

版权注意事项：1、书籍版权归著者和出版社所有；
2、本PDF仅用于个人获取知识，进行私底下知识交流；
3、PDF获得者不得在互联网以任何目的进行传播；
如有需要，请尽量购买正版实体书！支持书籍作者！！

Java

数字图像处理

编程技巧与应用实践

Digital Image Process in Java

贾志刚 著



机械工业出版社
China Machine Press

作者简介



贾志刚

CSDN博客专家，51CTO视频学院认证讲师，拥有10年以上的Java语言编程经验，在图像特征提取匹配、识别等方面有较深入的研究，多年从事Java Swing图形与图像方面的应用开发，拥有丰富的图像处理项目实践经验。



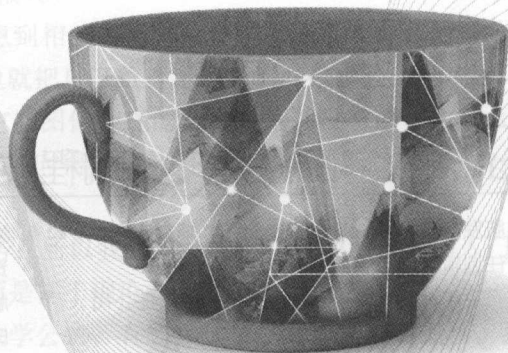
Java

数字图像处理

编程技巧与应用实践

Digital image process in Java

贾志刚 著



机械工业出版社
China Machine Press

图书在版编目 (CIP) 数据

Java 数字图像处理: 编程技巧与应用实践 / 贾志刚著. —北京: 机械工业出版社, 2015.11
(Java 核心技术系列)

ISBN 978-7-111-51946-1

I. J… II. 贾… III. JAVA 语言—程序设计 IV. TP312

中国版本图书馆 CIP 数据核字 (2015) 第 256250 号

Java 数字图像处理: 编程技巧与应用实践

出版发行: 机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码: 100037)

责任编辑: 余 洁 杨绣国

责任校对: 殷 虹

印 刷: 北京文昌阁彩色印刷有限责任公司

版 次: 2016 年 1 月第 1 版第 1 次印刷

开 本: 186mm×240mm 1/16

印 张: 21.75

书 号: ISBN 978-7-111-51946-1

定 价: 69.00 元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88379426 88361066

投稿热线: (010) 88379604

购书热线: (010) 68326294 88379649 68995259

读者信箱: hzit@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问: 北京大成律师事务所 韩光 / 邹晓东

Preface 前言

为什么要写这本书

我对图像处理的认识最初来自于读软件工程专业时做毕业设计论文的需要，毕业论文做完以后，我便把所有关于图像处理的知识扔到了一边。2011年的一天有位朋友问了我几个简单的图像处理方面的问题，在解答问题的过程中我发现自己对图像处理的热情一直都在燃烧，从那一刻起我决定重新学习图像处理。这之后，我把以前买的几本图像处理的书都读了一遍，同时还坚持通过写博客来督促自己加深理解，随着学习的不断深入，对图像处理的认知也在不断加深，我越来越渴望自己能实现那些书中提到的图像处理手段与方法，于是便开始不断尝试，在经过了各种“坑”与无助之后，我终于编程实现了学习过的每一种图像处理方法。这个过程十分痛苦，因为我深刻感受到了图像处理在理论与实践之间的细微差异，而这些细微差异往往会导致处理结果与理论预期相差很大。

可能提到图像处理，很多人马上就会想到相关书籍中各种复杂的数学公式与矩阵计算，然后就会说我数学不好学不了这个，早早地就把自己给否定了。那些数学公式的确让人望而生畏，但是只要仔细探究一番，就会发现它在图像处理的应用上远远没有看上去那么复杂，甚至可以说十分简单，这是本人学习图像处理时得到的最大一个心得体会，正如一句俗语说的：“世上无难事，只怕有心人”。

正是因为自己在学习过程中经历了痛苦，所以我想写一本不一样的图像处理入门图书，内容不再是冰冷的数学公式与文字描述，而是基于理论的实践步骤和细节详解，是一个可以直接运行的代码实现，书中没有大量的数学公式，有的只是数学知识的巧妙运用。我希望通过分享自己学习过程中的体会与编程实践经验，帮助更多人在学习图像处理的道路上少走弯路，早日进入图像处理的科学殿堂。

在国内，程序员写书早已不是什么新鲜事物，但是我可以肯定地说，本书是国内第一本由奋斗在编码一线的码农写的图像处理入门图书。它不是当下流行的视觉图像处理库的应用介绍，而是图像处理基础知识和理论的学习与实践，正如一句西方科技谚语所说的那样，“在理论上，理论与实践是一致的，在实践上，它们是不一致的”。当前关于图像处理的书大

多数都是重理论而轻实践，但图像处理在理论与实践编程之间是存在轻微差异的，而这就成了很多初学者无法逾越的鸿沟。本书就是要拟合理论与实践之间的鸿沟，帮助读者架起从理论到实践的大桥。

作为工作超过十年的程序员写的第一本书，本书也是我个人职业生涯的一个新起点，它鞭策与勉励自己不断努力上进，除了对图像处理的兴趣外，这一年多写书的动力更多的是毅力与帮助后来者的初衷。只要本书能为国内图像处理专业知识的普及与应用实践略尽绵薄之力，那辛苦也就值了。

读者对象

本书适合以下人群阅读：

- ❑ 从事图像处理的工作人员
- ❑ 学习图像处理的爱好者
- ❑ 希望提升自我的中高级程序员
- ❑ 计算机专业高年级本科生或研究生
- ❑ 开设图像处理相关课程的大专院校学生
- ❑ 从事 Java 应用的开发者

如何阅读本书

本书分为两大部分，其中第一部为前三章，主要介绍 Java Swing 编程的基础知识。第二部分是本书的核心内容，系统全面地介绍图像处理的各种方法与常见应用场景编程实现。如果你已经对 Java 语言和 Java Swing 有基本的认识，可以跳过前三章，直接从第 4 章开始阅读本书。同时本书注重实践，所以请务必阅读给出的源代码并运行它，这样才能更好地理解所讲的知识。

第一部分为基础篇，简单地介绍了 Java Swing 图形与图像编程基本 API 使用技巧，以及相关实践编程，帮助读者了解图像接口在 Java 语言中的基础知识，并熟悉像素的读写与操作。

第二部分为实践与应用编程，从最基础的像素操作开始，通过实践编程讲解图像处理过程中各种基本像素运算、混合、图像插值、直方图获取与图像搜索、图像卷积、边缘提取、二值图像分析与特征提取等知识，最后通过剖析一个流行的图像油画转换算法编程实践来结束本书。

附录为本书相关数学知识简单参考。其他参考资料索引可在我的 Github 上找到。

此外，本书的源文件可到 www.hzbook.com 上通过搜索本书下载，或者到 [github](https://github.com) 上下载。

勘误和支持

由于作者的水平有限，编写的时间也很仓促，书中难免会出现一些错误或不准确的地方，恳请读者批评指正。本书配套源代码已上传到 [github](https://github.com) 上，访问地址为：<https://github.com/>

gloomymyfish/mybook-java-imageprocess, 如果有读者想直接提交勘误之后的代码, 请先邮件联系本人, 同意以后即可提交, 同时本人也会根据读者反馈修改更新源代码。如果你有更多的宝贵意见, 也欢迎发送邮件至我的邮箱 bfnh1998@hotmail.com, 我很期待能够听到你们的真挚反馈。

致谢

首先要感谢图像处理学科那些伟大的先行者, 是他们创立了这个影响力巨大的学科。其次要感谢 CSDN 博客频道, 在 CSDN 我结识了很多良师益友, 他们直言不讳地指出了我博客文章中的很多不妥之处与需要改进的地方, 特别是 Trent、jichen324、qiwenmingshiwo、FDHGVH2461、cr459464757、wust 小吴、xiaowei_cqu, 以及这个仓促写就的名单之外的更多朋友, 感谢你们的宝贵建议。

感谢机械工业出版社华章公司的编辑杨绣国老师, 你的一封电子约稿邮件促成了本书, 也帮助我实现了写一本注重实践的图像处理入门图书的梦想; 感谢你的耐心, 在这一年多时间里你不厌其烦地回答我在写作过程中一个又一个问题; 感谢你的魄力和远见, 始终支持我的写作, 你的鼓励和帮助引导我顺利完成全部书稿。

最后一定要感谢我的父母, 感谢你们将我培养成人; 感谢我的妻子在我写书的这一年多时间让我从家务中解脱, 给我支持与鼓励。

谨以此书, 献给我最亲爱的两个孩子, 以及众多热爱图像处理的朋友们。

贾志刚

中国, 苏州, 2015 年 9 月

目 录 Contents

前言

第 1 章 Java Graphics 及其 API 简介 1

1.1 什么是 Java 图形设备 Graphics 1

1.1.1 Graphics 概述 2

1.1.2 Graphics 图形设备的获取、 使用和销毁 2

1.1.3 Java Swing Graphics2D 的重要 属性 3

1.2 Java 2D API 3

1.2.1 基本的 Java 2D 图形绘制 4

1.2.2 使用 Java 2D 实现太极图形 绘制 5

1.3 用 Java Swing 绘制自定义的 JPanel 6

1.4 Swing Java 2D 的其他高级特性 介绍 8

1.5 小结 13

第 2 章 Java BufferedImage 对象及其 支持的 API 操作 14

2.1 BufferedImage 对象的构成 14

2.1.1 Raster 对象的作用与像素存储 15

2.1.2 图像类型与 ColorModel 16

2.1.3 BufferedImage 对象的创建与 保存 17

2.1.4 一个完整的 ImageBuffered 读取 例子 18

2.2 Java BufferedImageOp API 20

2.2.1 Java BufferedImageOp 接口 介绍 20

2.2.2 BufferedImage 对象像素的读写 方法 21

2.2.3 常见问题举例 21

2.3 基于 BufferedImageOp 的图像滤镜 演示 22

2.4 小结 28

第 3 章 基本 Swing UI 组件与图像 显示 29

3.1 JPanel 组件与 BufferedImage 对象的显示 29

3.2 JFrame 组件与 Main UI 实现 31

3.3 JFileChooser 文件选择框的使用 32

3.4 基本 JButton 事件响应 32

3.5 一个完整的 Swing UI Demo 33

3.6 小结 37

第4章 图像属性	39	7.2 临近点插值算法.....	117
4.1 失去的时光与回忆——老照片特效.....	39	7.3 双线性插值算法.....	120
4.2 图像属性.....	42	7.4 双立方插值与 Lanczos 采样.....	124
4.3 图像的亮度、对比度和饱和度.....	45	7.4.1 双立方插值算法.....	124
4.4 图像饱和度调整.....	46	7.4.2 Lanczos 采样插值算法.....	131
4.5 图像亮度调整.....	50	7.5 图像旋转.....	134
4.6 图像对比度调整.....	53	7.6 小结.....	141
4.7 综合应用——调整图像亮度、对比度和饱和度.....	55	第8章 图像卷积	143
4.8 小结.....	61	8.1 模糊也是一种美.....	143
第5章 像素基本操作	62	8.2 图像空间域卷积.....	145
5.1 大自然的色彩——自然系列滤镜.....	62	8.3 盒子模糊与高斯模糊.....	149
5.2 图像像素加减乘除.....	65	8.3.1 盒子模糊.....	150
5.3 两幅图像的融合与叠加.....	70	8.3.2 高斯模糊.....	154
5.4 一个更加深入的应用实践——图像上轧花文字效果.....	75	8.4 边缘保留的模糊算法——高斯双边模糊.....	157
5.5 小结.....	82	8.5 像素格特效.....	163
第6章 像素统计与应用	83	8.6 卷积应用：图像去噪.....	165
6.1 统计图像的均值、最大值与最小值.....	83	8.7 图像锐化、拉普拉斯滤波.....	173
6.2 灰度图像二值化.....	86	8.8 小结.....	176
6.3 图像直方图.....	91	第9章 边缘检测与提取	177
6.4 基于直方图实现图像二值化.....	96	9.1 什么是图像的边缘.....	177
6.5 应用——直方图均衡化.....	100	9.2 Robot 算子与轧花效果.....	179
6.6 应用——基于直方图的图像搜索.....	105	9.3 Sobel 算子与 Prewitt 算子.....	182
6.7 小结.....	109	9.4 图像梯度——大小与角度.....	186
第7章 图像编辑	110	9.5 基于二阶导数的图像边缘提取.....	189
7.1 为什么图像放大以后失真.....	110	9.6 经典边缘提取算法——Canny Edge Detection.....	193
		9.7 小结.....	200
		第10章 二值图像	201
		10.1 二值图像概述与半色调算法.....	201

10.2	图像抖动算法	204
10.3	二值图像泛洪填充算法	208
10.4	连通组件标记算法	212
10.5	二值图像边缘跟踪	218
10.6	二值图像细化	224
10.7	计算连通区域几何质心	228
10.8	计算连通区域方向角度	231
10.9	小结	233

第 11 章 图像形态学 235

11.1	像素集合操作	235
11.2	腐蚀与膨胀	238
11.3	开闭操作	241
11.4	Hit-and-Miss 变换操作	244
11.5	距离变换	247
11.6	分水岭算法	250
11.7	灰度图像腐蚀与膨胀	254
11.8	小结	257

第 12 章 图像分割 258

12.1	抠图真的这么难吗	258
12.2	基于 Mean-Shift 的图像分割	259
12.3	基于 K-Means 的图像分割	265

12.4	基于 Fuzzy C-Means 的图像分割	269
12.5	基于分水岭的图像分割	275
12.6	小结	279

第 13 章 图像特征的提取与检测 280

13.1	颜色特征提取	280
13.2	纹理提取	283
13.3	直线检测	288
13.4	圆检测	291
13.5	图像金字塔	295
13.6	Harris 角度检测	302
13.7	SIFT 特征提取	307
13.8	小结	322

第 14 章 综合运用：照片转油画 323

14.1	画笔区域	323
14.2	采样问题	325
14.3	笔画参数	327
14.4	笔画绘制	330
14.5	程序实现	334
14.6	小结	337

附录 数学知识参考引用 338

Java Graphics 及其 API 简介

在开始本书内容之前，笔者假设你已经有了面向对象语言编程的基本概念，了解 Java 语言的基本语法与特征，原因在于本书的所有源代码都是基于 Java 语言实现的，而且是基于 Java 开发环境运行与演示所有图像处理算法的。本书第 1 章到第 3 章是为了帮助读者了解与掌握 Java 图形与 GUI 编程的基本知识与概念而写的。本章主要介绍 Java GUI 编程中基本的图形知识，针对 GUI 编程，Java 语言提供了两套几乎并行的 API，分别是 Swing 与 AWT。早期的 Java GUI 编程中主要使用 AWT 的相关组件，但是 AWT 的功能并不是十分强大，而且严重依赖本地接口。于是在 Java 1.3 及后续版本中引入了 Swing 工具实现 GUI 编程，Swing 中的组件大多数都是基于纯 Java 语言实现的，而不是通过本地组件实现的，所以它们是轻量级的 GUI 组件，同时 Swing 对图形与图像的支持操作也有很大的提高与增强。如何区分 AWT 组件与 Swing 组件？一个简单而且相当直观的方法是看 Class 的名称，Swing 的组件大多数带有大写的前缀字母 J。

Graphics 作为 Java 的图形引擎绘制接口，几何形状、文字、图像的绘制都必须通过它完成，此外，Graphics 还支持绘制过程的控制，可以设置画笔颜色、纹理、颜色填充方法、合成与裁剪路径及各种 Stroke 与 Fill 的属性等。用户程序通常都是通过 Graphics 来访问绘制引擎，从而实现各种图形与图像绘制的，因此可以说 Graphics 是 Swing 中最重要的接口对象。好吧，下面让我们一起揭开 Graphics 的神秘面纱。

1.1 什么是 Java 图形设备 Graphics

简单地讲 Graphics 是 Java 图形绘制引擎的访问接口，只有通过它才可以访问到 Java GUI

的图形绘制引擎，实现图形的绘制与绘制过程的控制。

1.1.1 Graphics 概述

Graphics 的功能大致可以分为两类，第一类是通过 Draw 或 Fill 方法来实现各种图形的绘制与填充，第二类是设置各种绘制属性，最简单的包括设置字体、颜色、填充方法等。此外，在 Java 2D 中 Graphics 还可以被转型为 Graphics2D 对象，从而提供更高精度的图形绘制，设置更多绘制属性来控制绘制过程。

1.1.2 Graphics 图形设备的获取、使用和销毁

在 Java Swing 中正确获取 Graphics 对象的方法有两种。

第一种是从 BufferedImage 对象实例中获取，其代码如下：

```
bufferedImage.createGraphics() // bufferedImage 为对象实例
```

第二种方法是通过重载 Swing 组件的 paintComponent (Graphics g) 或 paint (Graphics g) 方法来实现，个人推荐采用重载 paintComponent (Graphics g) 方法来实现，因为重载 paint (Graphics g) 是 AWT 时代遗留下来的产物，是一个重量级绘制重载，通常用于 Canvas 对象的重载绘制。

除了以上两种推荐的做法以外，笔者经常还看到直接通过 Swing 组件的 getGraphics 去获取 Graphics 对象的，这样做的坏处是一旦该组件没有被显示，所对应的 Graphics 对象将返回 NULL。而且这种做法常会导致一些意想不到的错误，所以应该尽量避免这么做。下面提供一个这么做导致错误的代码示例，如下：

```
JButton okBtn = new JButton("OK");
okBtn.getGraphics().drawRect(0,0,20,20); // NullPointerException
```

在获取了 Java 图形设备对象 Graphics 之后，就可以调用它的绘制方法来实现图形绘制与填充了。简单的示例代码如下：

```
public void paintComponent(Graphics g)
{
    g.setColor(Color.BLUE);
    g.drawRect(10, 10, 50, 50);
}
```

上述代码将会绘制一个蓝色边框的矩形，其中“10, 10”表示矩形开始绘制的左上角位置，“50, 50”分别代表矩形的长度与高度。

在使用完 Graphics 对象以后，请记住一定要销毁图形设备对象，可通过调用方法 dispose() 来释放图形绘制时所使用的任何资源。特别是当图形设备是从你自己的 BufferedImage 对象中创建出来的时候，记得使用完以后一定要调用 dispose() 方法来释放资源。假设没有调用 dispose()，一般情况下 Java 的 GC 也会自动调用来释放资源，但还是强烈建议在绘

制完成以后显式调用 `dispose()` 方法来确保被使用的资源得到及时释放而不是依赖 Java GC。原因在于当你使用的 `Graphics` 来自 `BufferedImage` 对象时, `Graphics` 对象不会被自动销毁, 而依赖 GC 调用来清理与释放资源并不能保证及时释放, 可能导致程序堆内存过度消耗产生 OOM (Out of Memory) 问题。

1.1.3 Java Swing Graphics2D 的重要属性

`Graphics` 可以向下转型为 `Graphics2D` 对象, 可以通过设置绘制属性来实现对图形绘制质量的控制。其接受对象为 `RenderingHints` 的枚举类型, 通过方法 `setRenderingHint (RenderingHints.key, RenderingHints.value)` 来实现, 一般常用的 Key 与 Value 有如下形式。

控制图形边缘反锯齿时, `RenderingHints.KEY_ANTIALIASING` 的值为:

- ☐ `RenderingHints.VALUE_ANTIALIAS_ON` 表示支持边缘反锯齿。
- ☐ `RenderingHints.VALUE_ANTIALIAS_OFF` 表示不支持边缘反锯齿。

控制文字或文本边缘反锯齿时, `RenderingHints.KEY_TEXT_ANTIALIASING` 的值为:

- ☐ `RenderingHints.VALUE_TEXT_ANTIALIAS_ON` 表示支持文本边缘反锯齿。
- ☐ `RenderingHints.VALUE_TEXT_ANTIALIAS_OFF` 表示不支持文本边缘反锯齿。

控制图像的插值方法时, `KEY_INTERPOLATION` 的值为:

- ☐ `RenderingHints.VALUE_INTERPOLATION_BICUBIC` 表示使用双立方插值方法。
- ☐ `RenderingHints.VALUE_INTERPOLATION_BILINEAR` 表示使用双线性插值方法。
- ☐ `RenderingHints.VALUE_INTERPOLATION_NEAREST_NEIGHBOR` 表示使用临近点插值方法。

控制绘制方法时, `KEY_RENDERING` 的值为:

- ☐ `RenderingHints.VALUE_RENDER_QUALITY` 表示支持绘制质量优先。
- ☐ `RenderingHints.VALUE_RENDER_SPEED` 表示支持绘制速度优先。

控制绘制过程是否支持抖动时, `KEY_DITHERING` 的值为:

- ☐ `RenderingHints.VALUE_DITHER_DISABLE` 表示不支持抖动。
- ☐ `RenderingHints.VALUE_DITHER_ENABLE` 表示支持抖动。

更多的绘制属性控制可以参考官方文档, 需要强调的是, 由于 Java 的跨平台属性导致并不是所有的 `RenderingHints` 设置都会起作用, 因此有些属性可能只有在某些特定的平台才支持。但是最常见的图形与文本的反锯齿功能几乎所有的操作系统平台都支持!

1.2 Java 2D API

当 `Graphics` 向下转型为 `Graphics2D` 时, Java 2D 的图形绘制引擎得以访问, 一个功能更加丰富的图形库呈现在读者眼前, 它就是 Java 2D API。如果你问笔者 Java 2D 与 Swing 有何关系, 可以很认真地说, 二者毫无瓜葛, Java 通过引入 Swing、Java 2D 与 Java 3D, 极大地

丰富了 Java 的图形功能，使应用程序接口更加完善，为各种可能的图形开发提供了可靠保证与全面支持，从而也使得学习 Java 图形方面的知识时不再那么无助了。下面来看一下 Java 2D 对图形支持与改进都包括了哪些：

❑ 为显示设备与打印机提供统一的绘制引擎。

❑ 一个广泛的几何形状支持。

❑ 文档打印支持。

❑ 可控制的绘制质量。

❑ 增强的色彩支持。

❑ 文字、形状、图像绘制检测。

1.2.1 基本的 Java 2D 图形绘制

Java 2D 图形绘制支持的图形形状如图 1-1 所示。

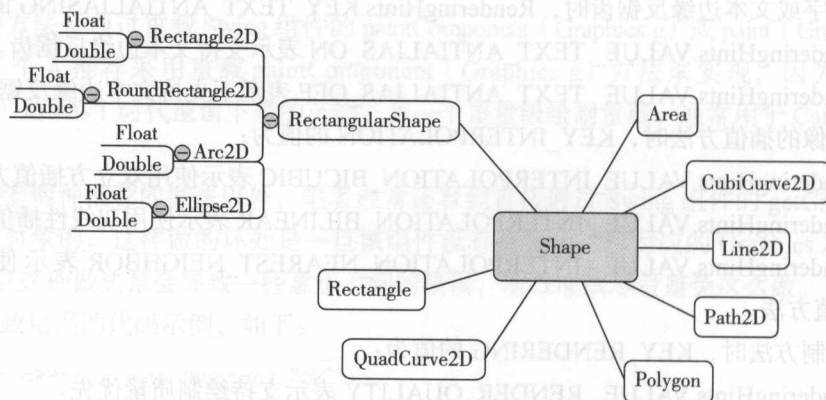


图 1-1 Java2D 形状绘制支持接口

Java 2D 图形绘制最常见的是将绘制代码放在 `paintComponent (Graphics g)` 方法中，显示时 Swing 会首先调用 `paint()` 方法。该方法会调用下面的三个方法：

❑ `paintComponent (Graphics g)`

❑ `paintBorder (Graphics g)`

❑ `paintChildren (Graphics g)`

在绝大多数情况下，图形绘制只需要重载 `paintComponent()` 方法来实现。一个基本图形绘制代码如下：

```

public void paintComponent(Graphics g)
{
    Graphics2D g2d = (Graphics2D)g;
    g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
        RenderingHints.VALUE_ANTIALIAS_ON); //反锯齿
}
  
```

```

g2d.setPaint(Color.BLUE);           // 设置画笔颜色
g2d.drawRect(10, 10, 50, 50);       // 绘制矩形
g2d.dispose();                       // 释放资源
}

```

1.2.2 使用 Java 2D 实现太极图形绘制

太极在中国源远流长，黑白相间的太极图案已经是一个文化标志，这里将使用 Java 2D 的图形绘制技术实现太极图案的绘制。如果仔细观察太极图案，就会发现它是非常精准的黑白对称图案。可通过设置画笔颜色来实现黑白颜色控制，利用 Java 2D Area 对图形布尔操作的支持实现太极图形绘制。Java 2D Area 对图形 Shape 对象进行支持的四种布尔操作如下。

- Union (加操作): 保留两个几何形状及其重叠部分。
- Subtraction (减操作): 从第一个几何形状减去与第二个几何形状重叠的部分。
- Intersection (可以看成与操作): 只保留两个几何形状重叠的部分。
- Exclusion-or (XOR 异或操作): 保留两个几何形状不重叠的部分。

这四种操作的示意图如图 1-2 所示。

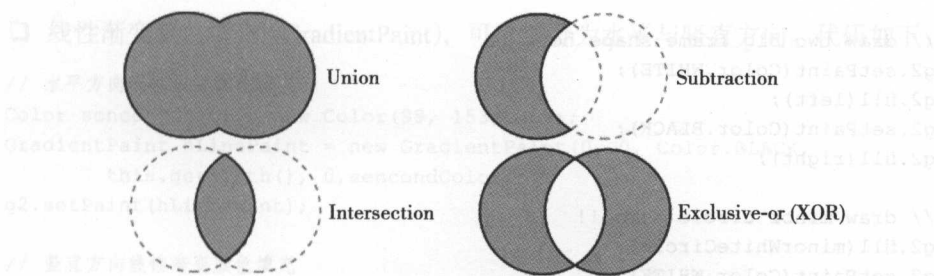


图 1-2 Java 2D 支持的四种几何操作

实现太极图案的相关代码如下：

```

protected void paintComponent(Graphics g) {
    Graphics2D g2 = (Graphics2D)g;
    g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
        RenderingHints.VALUE_ANTIALIAS_ON);
    // R = 150
    Shape lefthalfCircle = new Ellipse2D.Double(10,10, 300,300);
    // R = 150
    Shape righthalfCircle = new Ellipse2D.Double(10,10, 300,300);
    // R/2 = 75
    Shape innerCircle1 = new Ellipse2D.Double(85,10, 150,150);
    // R = 150
    Shape innerCircle2 = new Ellipse2D.Double(85,160, 150,150);
    Shape rectangel1 = new Rectangle2D.Double(160, 10, 150, 300);
    Shape rectangel2 = new Rectangle2D.Double(10, 10, 150, 300);

    Area left = new Area(lefthalfCircle);

```

```

Area right = new Area(righthalfCircle);

Area area11 = new Area(rectangel1);
Area area22 = new Area(rectangel2);

left.subtract(area11);
right.subtract(area22);

Area inner1 = new Area(innerCircle1);
Area inner2 = new Area(innerCircle2);

left.add(inner1);
right.add(inner2);

// trick is here !!!
right.subtract(inner1);

// create minor circle here!!! // ++ 60, R = 150
Shape minorWhiteCircle = new Ellipse2D.Double(150,70, 20,20);
Shape innerBlackCircle = new Ellipse2D.Double(150,230, 20,20);

// draw two big frame shape here...
g2.setPaint(Color.WHITE);
g2.fill(left);
g2.setPaint(Color.BLACK);
g2.fill(right);

// draw minor circle here!!!
g2.fill(minorWhiteCircle);
g2.setPaint(Color.WHITE);
g2.fill(innerBlackCircle);
}

```

运行源文件中第 1 章中的完整代码可以看到一个标准的太极图案。



注意 书中所有完整的源代码均已打包上传至 www.hzbook.com 和 [github](https://github.com) ^①，下载后按章节索引即可找到相应的代码，强烈建议运行每个源代码实例，将源代码看成本书的一部分。

1.3 用 Java Swing 绘制自定义的 JPanel

Swing 的 JPanel 组件是 GUI 编程中最重要的面板组件，可以通过重写 JPanel 中 paint-Component 方法实现对 JPanel 面板组件的背景颜色的调整或添加背景图片，进而实现自定义

① <https://github.com/gloomyfish/mybook-java-imageprocess>。

版本的面板 (JPanel) 组件。只要完成如下几步就可以实现一个简单自定义 JPanel 面板的绘制。

1) 实现对 JPanel 面板的继承, 代码如下:

```
public class CustomJPanel extends JPanel
{
    // 更多代码
}
```

2) 完成对 paintComponent (Graphics g) 对象的重载, 代码如下:

```
Protected void paintComponent(Graphics g)
{
    // 绘制代码
}
```

3) 访问 Graphics 绘制引擎, 设置画笔颜色并完成绘制, 在 Java 2D 中 paint 支持三种不同的画笔颜色填充策略, 它们分别是:

□ 单一颜色填充, 如 Color.BLUE、Color.RED 等。代码如下:

```
// 单一颜色背景填充
g2.setPaint(Color.BLUE);
```

□ 线性渐变颜色填充 (GradientPaint), 可以细分为水平与竖直方向。代码如下:

```
// 水平方向线性渐变颜色填充
Color sencondColor = new Color(99, 153, 255);
GradientPaint hLinePaint = new GradientPaint(0, 0, Color.BLACK,
    this.getWidth(), 0, sencondColor);
g2.setPaint(hLinePaint);
```

```
// 竖直方向线性渐变颜色填充
Color controlColor = new Color(99, 153, 255);
GradientPaint vLinePaint = new GradientPaint(0, 0, Color.BLACK,
    0, getHeight(), controlColor);
g2.setPaint(vLinePaint);
```

□ 圆周径向渐变颜色填充 (RadialGradientPaint), 支持两种以上的颜色渐变。代码如下:

```
// 圆周径向渐变颜色填充
float cx = this.getWidth() / 2;
float cy = this.getHeight() / 2;
float radius = Math.min(cx, cy);
float[] fractions = new float[] {0.1f, 0.5f, 1.0f};
Color[] colors = new Color[] {Color.RED, Color.GREEN,
    Color.BLUE};
// cx, cy表示圆周的中心点距离
// radius 表示半径长度,
// fractions表示色彩渐变关键帧位置, 每个值取值在0~1之间
// colors 表示颜色数组
RadialGradientPaint rgp = new RadialGradientPaint(cx, cy,
    radius, fractions, colors, CycleMethod.NO_CYCLE);
g2.setPaint(rgp);
```

4) 设置背景图片支持。很多时候我们希望 JPanel 背景是一张图片而不是颜色填充, 此时只需要将 BufferedImage 对象通过 drawImage() 方法放在 paintComponent() 中即可, 唯一需要注意的地方就是确保 BufferedImage 对象不为 NULL。代码如下:

```
// 图片作为背景填充
if(image != null)
{
    // 0, 0表示图像起始位置, 相对于坐标为左上角位置
    g2.drawImage(image, 0, 0, getWidth(), getHeight(), null);
}
```

5) 实现一个测试的 main 方法代码如下:

```
public static void main(String[] args)
{
    JFrame ui = new JFrame("Custom JPanel");
    ui.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    ui.getContentPane().setLayout(new BorderLayout());
    ui.getContentPane().add(new CustomJPanel(),
        BorderLayout.CENTER);
    ui.setPreferredSize(new Dimension(380, 380));
    ui.pack();
    ui.setVisible(true);
}
```

读者可以下载相关文档查看完整的源代码, 源代码是本书的一部分, 请读者尽量运行源代码, 这样可以更好地帮助读者理解所学内容。

1.4 Swing Java 2D 的其他高级特性介绍

1. Stroke 接口

Stroke 是 Graphics2D 的 API 接口, 用来实现图形的描边修饰, 在 Java 2D 中只有一个完成 Stroke 接口的类 BasicStroke, 如果有需要, 可以自己完成 Stroke 接口, 实现自定义的 Stroke 类。如何使用 Stroke 的实现类? 方法如下:

- 1) 调用 Graphics2D 的 setStroke() 方法, 传入一个实例化的 Stroke 对象。
- 2) 调用 draw() 方法, 传入要绘制的几何形状。

BasicStroke 的对象构造函数代码如下:

```
// 创建Stroke对象实例
float[] dash = {10.0f, 5.0f, 3.0f};
Stroke dashed = new BasicStroke(2.0f, BasicStroke.CAP_BUTT,
    BasicStroke.JOIN_MITER, 10.0f, dash, 0.0f);
```

其中:

- 第一个参数 2.0f 表示 Stroke 的宽度。

- ❑ 第二个参数声明 Stroke 的结束方式, `BasicStroke.CAP_BUTT` 表示如果不是闭合区域则不做任何修饰, 直接结束绘制, `BasicStroke.CAP_ROUND` 表示如果不是闭合则添加圆角帽线, 然后结束。
- ❑ 第三个参数表示线的连接方式, 此处为 `JOIN_MITER`。
- ❑ 第四个参数指定 Stroke 线段的长度, 此处线段长度为 10。
- ❑ 第五个参数声明点线模式, 此处点线模式 `dash` 为不等长线段。
- ❑ 第六个参数声明位移, 0.0 表示位移间隔为零。

更详细的参数说明可以参考 JDK 的官方文档, 下面的代码通过创建 `BasicStroke` 实例对象来绘制一个虚线矩形:

```
// 创建Stroke对象实例
float[] dash = {10.0f, 5.0f, 3.0f};
Stroke dashed = new BasicStroke(2.0f, BasicStroke.CAP_SQUARE,
    BasicStroke.JOIN_MITER, 10.0f, dash, 0.0f);

// 设置Graphics2D的Stroke对象引用
g2.setStroke(dashed);

// 创建形状
Shape rect2D = new RoundRectangle2D.Double(50, 50,
    300, 100, 10, 10);
g2.draw(rect2D);
```

2. Texture Fill 接口

Texture Fill 即纹理填充, `Graphics2D` 提供了 `setPaint()` 方法来设置纹理填充, 通过 `fill()` 方法可实现对几何形状的填充。前面讲到的两种填充方式分别为颜色填充与渐变填充, 这里将重点介绍纹理填充的类 `TexturePaint` 创建与使用。

`TexturePaint` 通过构造一个 `BufferedImage` 对象作为纹理来填充几何形状, 因为 `BufferedImage` 对象数据将被拷贝到 `TexturePaint` 中, 所以 `BufferedImage` 对象设置得比较小为好。实例化一个 `TexturePaint` 对象的代码如下:

```
Rectangle2D rect = new Rectangle2D.Double(10,10,200,200);
TexturePaint tp = new TexturePaint(image, rect)
```

其中 `image` 表示一个 `BufferedImage` 实例, `rect` 表示截取作为纹理的区域。

使用实例化的 `TexturePaint` 来完成对矩形区域填充的代码如下:

```
// Texture Fill
Rectangle2D rect = new Rectangle2D.Double(10,10,200,200);
TexturePaint tp = new TexturePaint(image, rect);
g2.setPaint(tp);
g2.fill(rect2D);
```

3. Font 属性

Java 2D 支持绝大多数常见字体的创建与属性值的修改调整，可通过 Graphics2D setFont() 方法来实现绘制字体的修改，同时 Graphics2D 绘制引擎还支持自定义的外部字体文件 *.ttf 的动态加载与使用。只要在使用之前加载字体文件即可，使用下面的代码可实现字体文件加载：

```
public Font loadFont() throws FontFormatException, IOException
{
    String fontFileName = "AMERSN.ttf";
    InputStream is = this.getClass().
        getResourceAsStream(fontFileName);
    Font actionJson = Font.createFont(Font.TRUETYPE_FONT, is);
    Font actionJsonBase = actionJson.deriveFont(Font.BOLD, 16);
    return actionJsonBase;
}
```

字体加载与使用的完整代码如下：

```
package com.book.chapter.one;

import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.FontFormatException;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.RenderingHints;
import java.io.IOException;
import java.io.InputStream;

import javax.swing.JFrame;
import javax.swing.JPanel;

public class FontDemo extends JPanel {

    private static final long serialVersionUID = 1L;

    public FontDemo() {
        super();
    }

    public void paintComponent(Graphics g) {
        Graphics2D g2d = (Graphics2D) g;
        // 反锯齿
        g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
            RenderingHints.VALUE_ANTIALIAS_ON);
        // 设置画笔颜色
        g2d.setPaint(Color.BLUE);
        try {
```

```

        g2d.setFont(loadFont());
    } catch (FontFormatException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
    g2d.drawString("Font Demo", 50, 50);
    g2d.dispose(); // 释放资源
}

public Font loadFont() throws FontFormatException,
IOException {
    String fontFileName = "AMERSN.ttf";
    InputStream is = this.getClass().
        getResourceAsStream(fontFileName);
    Font actionJson = Font.createFont(
        Font.TRUETYPE_FONT, is);
    Font actionJsonBase =
        actionJson.deriveFont(Font.BOLD, 16);
    return actionJsonBase;
}

public static void main(String[] args) {
    JFrame ui = new JFrame("Font Demo Graphics2D");
    ui.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    ui.getContentPane().setLayout(new BorderLayout());
    ui.getContentPane().add(new FontDemo(),
        BorderLayout.CENTER);
    ui.setPreferredSize(new Dimension(380, 380));
    ui.pack();
    ui.setVisible(true);
}
}

```

4. GeneralPath 与自定义几何形状

Java 2D 支持通过 GeneralPath 实现绘制任意的几何形状，使用 GeneralPath 提供的 API 接口绘制几何形状的步骤大致如下：

- 1) 实例化 GeneralPath 对象。
- 2) 调用 moveTo() 方法锚地开始点坐标。
- 3) 调用 lineTo() 或 curveTo 方法绘制连接线。
- 4) 调用 closePath() 方法完成几何形状绘制。

下面的代码实现了利用 GeneralPath 对象绘制一个红色五角星图案。

```

package com.book.chapter.one;

import java.awt.BorderLayout;
import java.awt.Color;

```

```

import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.RenderingHints;
import java.awt.geom.GeneralPath;

import javax.swing.JFrame;
import javax.swing.JPanel;

public class GeneralPathDemo extends JPanel {
    /**
     *
     */
    private static final long serialVersionUID = 1L;

    public GeneralPathDemo() {
        super();
    }

    public void paintComponent(Graphics g) {
        Graphics2D g2d = (Graphics2D) g;
        // 反锯齿
        g2d.setRenderingHint(
            RenderingHints.KEY_ANTIALIASING,
            RenderingHints.VALUE_ANTIALIAS_ON);

        // 五角星的五个点坐标
        int x1 = this.getWidth() / 2;
        int y1 = 20;
        int x2 = this.getWidth() / 5;
        int y2 = this.getHeight() - 20;
        int x3 = x2 * 4;
        int y3 = this.getHeight() - 20;
        int x4 = 20;
        int y4 = this.getHeight() / 3;
        int x5 = this.getWidth() - 20;
        int y5 = y4;

        // 定义画点的顺序
        int x1Points[] = { x1, x2, x5, x4, x3 };
        int y1Points[] = { y1, y2, y5, y4, y3 };

        // 设置填充颜色
        g2d.setPaint(Color.RED);

        // 实例化GeneralPath对象
        GeneralPath polygon = new GeneralPath(
            GeneralPath.WIND_EVEN_ODD,
            x1Points.length);
        // 锚地开始第一个点
        polygon.moveTo(x1Points[0], y1Points[0]);

```



```

// 顺序画出剩下点
for (int i = 1; i < x1Points.length; i++) {
    polygon.lineTo(x1Points[i], y1Points[i]);
}

// 调用closePath形成一个封闭几何形状
polygon.closePath();

// 绘制它
g2d.draw(polygon);

// 释放资源
g2d.dispose();
}

public static void main(String[] args) {
    JFrame ui = new JFrame("Demo Graphics");
    ui.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    ui.getContentPane().setLayout(new BorderLayout());
    ui.getContentPane().add(new GeneralPathDemo(),
        BorderLayout.CENTER);
    ui.setPreferredSize(new Dimension(380, 380));
    ui.pack();
    ui.setVisible(true);
}
}

```

1.5 小结

作为全书的第一章，本章主要介绍了 Java Swing 中关于图形 GUI 支持的一些基本概念与知识。从 Graphics2D 图形绘制引擎访问接口入手，首先介绍了 Graphics2D 的使用、属性设置和基本功能，接着介绍了 Graphics2D 图形包 Java 2D 的基本 API 的使用方法，以及几何图形之间的布尔操作等知识，并以太极图案为实例说明了几何图形之间布尔操作的用法，然后介绍了如何重载 Swing 非顶层组件 JPanel 的 paintComponent() 方法实现对 JPanel 面板自定义颜色填充，以及 Java 2D 中颜色填充常用的对象使用方法，帮助读者更好地理解使用 Java 进行图形绘制的基本方法与步骤。最后介绍了在 Graphics2D 中如何设置字体，设置 Stroke 风格，设置背景填充纹理等高级知识点，此外，还给出了 GeneralPath 对象的使用方法，实现了任意几何形状的绘制。

Graphics2D 作为绘制引擎接口，已经提供了当前编程语言图形包中所能提供的几乎所有功能，常见图形包中都有 draw 与 fill，都会用到 Texture、Stroke 等属性设置，都会涉及字体加载与使用，但是本章并没有对 Graphics2D 中图形错切、旋转与放缩、几何图形的透明混合规则等进行探讨，感兴趣的读者可以自己去做更进一步的研究。

Java BufferedImage 对象及其支持的 API 操作

第 1 章我们一起学习了 Java 中的 Graphics 图形包基本概念与知识，本章将介绍 Java 中关于图像文件操作的基本知识。首先是 Java 2D 图像对象 BufferedImage 的组件构成、与图像文件之间的关系、格式支持，以及如何利用 BufferedImage 对象在 Java 语言中实现像素读写操作。然后通过 BufferedImageOp 接口介绍 Java 中几种非常有用的对像素操作的 BufferedImageOp 的实现类。最后将集合上述知识点，实现一个简单 Java Swing 的滤镜程序，帮助读者实现学以致用，加深理解。

在介绍本章内容之前，笔者假设你已经掌握了基本 Java 语言编程知识，学习过简单的 Swing 程序，同时对图像文件的格式及其特点有一些简单的了解。这些知识点可以帮助你更好地学习本章内容。

2.1 BufferedImage 对象的构成

BufferedImage 是一个内存对象，当通过 ImageIO.read() 方法读取一个图像文件时，读取到的关于图像文件的所有信息都会被存储在该 API 返回的 BufferedImage 内存对象中。此外还可以通过 BufferedImage 类的构造函数来创建 BufferedImage 内存对象。BufferedImage 对象中最重要的两个组件为 Raster 与 ColorModel，分别用于存储图像的像素数据与颜色数据，BufferedImage 中的其他属性还包括宽、高、图像类型等。当需要对 BufferedImage 对象实现一些像素级别的操作时，调用 Raster 对象总是有点道理，如果做个形象的比喻，Raster 就好像一个像素操作的场地，任何像素读写操作都可以通过调用 Raster 相关接口来完成。一个完整的 BufferedImage 构成类关系图如图 2-1 所示。

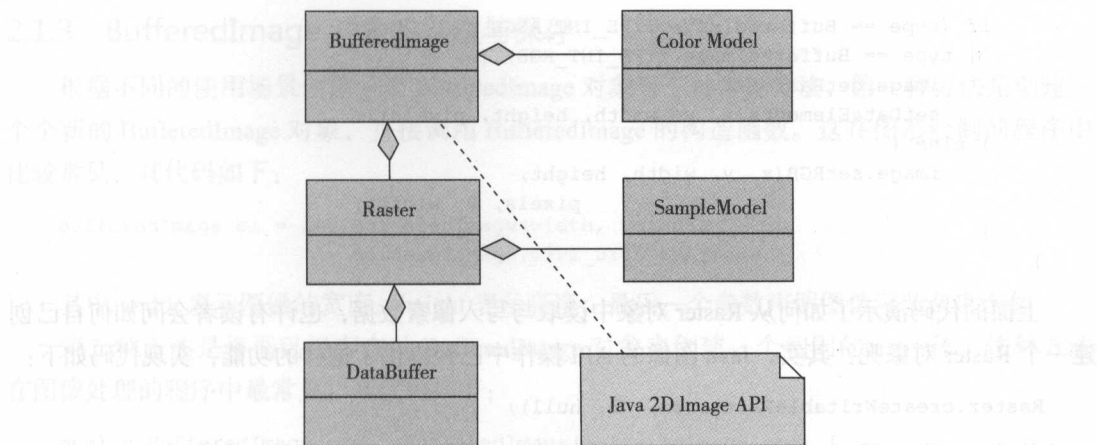


图 2-1 BufferedImage 对象关系图

2.1.1 Raster 对象的作用与像素存储

由于 Raster 对象是 BufferedImage 对象中的像素数据存储对象，因此，BufferedImage 支持从 Raster 对象中获取任意位置 (x, y) 点的像素值 $p(x, y)$ 。对于任意的 BufferedImage 对象来说，拥有越多的像素，Raster 对象需要的内存空间也就越大，同时 Raster 对象需要的内存空间的大小还跟每个像素需要存储的字节数有一定的关系。首先来探讨一下如何从 Raster 对象获取像素数据，从 Raster 对象中读取 BufferedImage 全部像素数据的代码如下：

```

public int[] getRGB(BufferedImage image, int x, int y,
    int width, int height, int[] pixels) {
    int type = image.getType();
    if (type == BufferedImage.TYPE_INT_ARGB
        || type == BufferedImage.TYPE_INT_RGB) {
        return (int[]) image.getRaster().
            getDataElements(x, y, width, height, pixels);
    } else {
        return image.getRGB(x, y, width,
            height, pixels, 0, width);
    }
}

```

上述方法实现了从 Raster 中读取像素数据，其中 x, y 表示开始的像素点，width 与 height 表示像素数据的宽度与高度，pixels 数组用来存放获取到的像素数据，image 是一个 BufferedImage 的实例化引用。向 BufferedImage 对象实例中写入像素数据需要通过 Raster 来完成，其代码如下：

```

public void setRGB(BufferedImage image, int x, int y,
    int width, int height, int[] pixels) {
    int type = image.getType();

```

```

if (type == BufferedImage.TYPE_INT_ARGB
|| type == BufferedImage.TYPE_INT_RGB) {
    image.getRaster().
        setDataElements(x, y, width, height, pixels);
} else {
    image.setRGB(x, y, width, height,
                pixels, 0, width);
}
}

```

上面的代码演示了如何从 Raster 对象中读取与写入像素数据，也许有读者会问如何自己创建一个 Raster 对象呢？其实，Java 图像的 API 操作中已经提供了这样的功能，实现代码如下：

```
Raster.createWritableRaster(sm, db, null);
```

其中 sm 指的是 SampleModel 对象实例，db 表示 DataBuffer 对象实例，最后一个参数 Point 参数默认为 null。如何创建 SampleModel 将在下一小节中详细解释。

2.1.2 图像类型与 ColorModel

从前面的内容可以知道，BufferedImage 对象中最重要的一个组件是 ColorModel 对象，最常用的实现类是 IndexColorModel，下面就以此为例来演示如何创建与使用 ColorModel 对象。首先来看如何创建一个 IndexColorModel 对象，IndexColorModel 的构造函数有五个参数，分别为：

- ❑ Bits：表示每个像素的所占的位数，对 RGB 单色来说是 8 位。
- ❑ Size：表示颜色组件数组长度，对于 RGB 取值范围 0 ~ 255 而言，值为 256。
- ❑ r[]：字节数组 r 表示颜色组件的 RED 值数组。
- ❑ g[]：字节数组 r 表示颜色组件的 GREEN 值数组。
- ❑ b[]：字节数组 r 表示颜色组件的 BLUE 值数组。

通常而言，每个单色所占的位数都在 1 ~ 16 之间，size 值必须大等于 1。正确创建 IndexColorModel 的代码如下：

```

public IndexColorModel getColorModel() {
    byte[] r = new byte[256];
    byte[] g = new byte[256];
    byte[] b = new byte[256];
    for (int i = 0; i < 256; i++)
    {
        r[i] = (byte) i;
        g[i] = (byte) i;
        b[i] = (byte) i;
    }
    return new IndexColorModel(8, 256, r, g, b);
}

```

BufferedImage 对象中最重要的两个组件如何创建我们都知道了，下面一小节就一起来看 BufferedImage 对象本身是如何创建的。

2.1.3 BufferedImage 对象的创建与保存

根据不同的使用场景创建一个 BufferedImage 对象有三种常见方法，第一种方法是创建一个全新的 BufferedImage 对象，直接调用 BufferedImage 的构造函数。这在图形绘制的程序中比较常见，其代码如下：

```
BufferedImage bi = new BufferedImage(width, height,
    BufferedImage.TYPE_BYTE_GRAY);
```

其中 width 表示图像的宽度，height 表示高度，最后一个参数声明图像字节灰度图像。

第二种方法是根据已经存在的 BufferedImage 对象来创建一个相同的 copy 体。这种方法在图像处理的程序中最常见，其代码如下：

```
public BufferedImage createBufferedImage(BufferedImage src) {
    ColorModel cm = src.getColorModel();
    BufferedImage image = new BufferedImage(cm,
        cm.createCompatibleWritableRaster(
            src.getWidth(),
            src.getHeight()),
        cm.isAlphaPremultiplied(), null);
    return image;
}
```

第三种方法是通过创建 ColorModel 与 Raster 对象实现 BufferedImage 对象的实例化，其代码如下：

```
public BufferedImage createBufferedImage(int width,
    int height,
    byte[] pixels) {
    ColorModel cm = getColorModel();
    SampleModel sm =
        getIndexSampleModel((IndexColorModel) cm,
            width, height);
    DataBuffer db = new DataBufferByte(pixels,
        width * height, 0);
    WritableRaster raster =
        Raster.createWritableRaster(sm, db, null);
    BufferedImage image = new BufferedImage(cm,
        raster, false, null);
    return image;
}
```

上述几种方法都是关于如何创建一个 BufferedImage 对象的，下面来看一下如何保存 BufferedImage 对象为本地图像文件。Java 中提供了 ImageIO 工具类来实现图像文件与 BufferedImage 对象之间的转换，读取一个图像文件时使用如下代码即可：

```
public BufferedImage readImageFile(File file)
{
    try {
```

```

        BufferedImage image = ImageIO.read(file);
        return image;
    } catch (IOException e) {
        e.printStackTrace();
    }
    return null;
}

```

保存 `BufferedImage` 对象为图像文件的代码如下：

```

public void writeImageFile(BufferedImage bi)
    throws IOException
{
    File outputfile = new File("saved.png");
    ImageIO.write(bi, "png", outputfile);
}

```

2.1.4 一个完整的 `ImageBuffered` 读取例子

本例将会演示前面所讲到的关于 `BufferedImage` 对象的所有知识点，包括像素的读取、`Raster` 对象的创建、`ColorModel` 的使用等。下面的代码演示了通过获取鼠标位置改变图像 `ColorModel` 对象索引，从而实现图像像素自动变化的方法。

```

package com.book.chapter.two;

import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.event.MouseEvent;
import java.awt.event.MouseMotionListener;
import java.awt.image.BufferedImage;
import java.awt.image.ColorModel;
import java.awt.image.DataBuffer;
import java.awt.image.DataBufferByte;
import java.awt.image.IndexColorModel;
import java.awt.image.Raster;
import java.awt.image.SampleModel;
import java.awt.image.SinglePixelPackedSampleModel;
import java.awt.image.WritableRaster;

```

```

import javax.swing.JFrame;
import javax.swing.JPanel;

public class BufferedImageDemo extends JPanel
    implements MouseMotionListener {

```

```

    private BufferedImage image = null;
    private int width = 350;
    private int height = 350;

```

```

public BufferedImageDemo() {
    image = createImage();
    addMouseListener(this);
}

@Override
public void mouseMoved(MouseEvent e) {
    // 创建新的图片，基于新的颜色模型索引
    image = new BufferedImage(createColorModel(e.getX()),
        image.getRaster(), false, null);
    repaint();
}

@Override
public void mouseDragged(MouseEvent e) {
}

public void paintComponent(Graphics g) {
    Graphics2D g2d = (Graphics2D) g;
    if(image != null) {
        g2d.drawImage(image, 2, 2,
            width, height, null);
    }
}

private BufferedImage createImage() {
    byte[] pixels = new byte[width * height];
    DataBuffer dataBuffer = new DataBufferByte(pixels,
        width*height, 0);
    SampleModel sampleModel = new
        SinglePixelPackedSampleModel(
            DataBuffer.TYPE_BYTE,
            width, height, new int[] {(byte)0xf});
    WritableRaster raster = Raster.createWritableRaster(
        sampleModel, dataBuffer, null);
    return new BufferedImage(createColorModel(0),
        raster, false, null);
}

private static ColorModel createColorModel(int n) {
    byte[] r = new byte[16];
    byte[] g = new byte[16];
    byte[] b = new byte[16];

    for (int i = 0; i < r.length; i++) {
        r[i] = (byte) n;
        g[i] = (byte) n;
        b[i] = (byte) n;
    }

    return new IndexColorModel(4, 16, r, g, b);
}

```

```

    }

    public static void main(String[] args) {

        JFrame ui = new JFrame("BufferedImage Demo");
        ui.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        ui.getContentPane().setLayout(new BorderLayout());
        ui.getContentPane().add(new BufferedImageDemo(),
                                BorderLayout.CENTER);
        ui.setPreferredSize(new Dimension(380, 380));
        ui.pack();
        ui.setVisible(true);
    }
}

```

2.2 Java BufferedImageOp API

本节将介绍 Java 中最常用的操作图像像素的 API 接口 BufferedImageOp，通过它，可以实现图像像素的调整，呈现出不同的图像显示效果，并且可编辑图像内容等。

2.2.1 Java BufferedImageOp 接口介绍

当前 BufferedImageOp 接口中最重要的方法是 filter() 方法，这是所有实现类必须完成的。目前 BufferedImageOp 有下面五个实现类。

- ❑ AffineTransformOp：主要操作是提供各种旋转、错切、放缩功能。
- ❑ ColorConvertOp：主要操作是提供像素灰度功能。
- ❑ ConvolveOp：主要用来实现图像卷积操作，可以实现模糊、边缘提取等效果。
- ❑ LookupOp：主要用来实现图像像素颜色的各种变换、反色等操作。
- ❑ RescaleOp：主要用来实现图像对比度与亮度的调整操作。

对应的类关系图如图 2-2 所示。

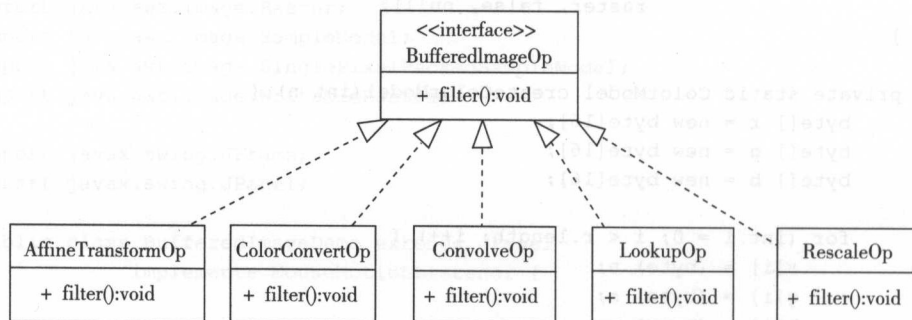


图 2-2 Java 2D 中 BufferedImageOp 接口实现类

2.2.2 BufferedImage 对象像素的读写方法

首先来看看如何读取一个像素的 RGB 值。读取一个像素点的 RGB 值的代码如下：

```
index = row * width + col;
ta = (pixels[index] >> 24) & 0xff;
tr = (pixels[index] >> 16) & 0xff;
tg = (pixels[index] >> 8) & 0xff;
tb = pixels[index] & 0xff;
```

其中 index 表示该像素点位置，row 表示纵坐标位置，col 表示横坐标位置。图像左上角的像素点对应位置坐标为 (0,0)，右下角的位置坐标为 (height-1, width-1)，这里的 height 又表示图像的高度，width 表示图像的宽度。

设置一个像素点的 RGB 值的代码如下：

```
index = row * width + col;
pixels[index] = (ta << 24) |
    (tr << 16) | (tg << 8) | tb;
```

其中 ta 表示透明度的值、tr 表示 RED 通道值、tg 表示 GREEN 通道值、tb 表示 BLUE 通道值，它们的取值范围为 [0 ~ 255]。



注意 当创建的 BufferedImage 对象为 ARGB 类型，使用 ImageIO 保存为 JPG 格式时，你可能会发现图像与在 Swing UI 中显示的不一样，色差很大，原因在于 ARGB 类型与 JPG 格式不是很兼容，选择 ARGB 时最好保存为 PNG 格式，而 RGB 格式保存为 JPG 格式。

2.2.3 常见问题举例

当笔者初次接触 Java Image API 时，在学习过程中遇到了各种各样的问题。下面整理总结一下初学者经常遇到的问题。

1) 图像格式支持。很多人会用 Java Image API 来读取 tiff 文件，可惜这种格式并不被支持，在 JDK6 中，Java ImageIO 类只支持几种常见的图像格式 (PNG、JPG、GIF、BMP)，其他均不支持。

2) 透明通道的支持与文件保存。如果 BufferedImage 对象为 ARGB 格式，说明支持透明通道，最佳的保存格式为 PNG 格式，否则会造成图像信息丢失，保存失真。

3) 加载图像资源文件。很多人不知道如何加载图像资源文件，笔者最喜欢的做法是把要使用的图像文件与 Class 文件放在同一个 package 中，然后通过如下代码加载：

```
java.net.URL imageURL = this.getClass().
    getResource("lena.jpg");
BufferedImage bi = ImageIO.read(imageURL);
```

4) 处理结果显示的像素里有大片的白点或黑白斑点，最可能的原因在于你的 RGB 像素

值取值范围超过了 [0 ~ 255]，请仔细检查。

5) 如何在 `BufferedImage` 对象上绘制各种几何形状与文字等信息？使用如下代码即可：

```
Graphics2D g2 = image.createGraphics(); // 获取绘制引擎
g2.draw(...);                          // 绘制几何形状Shape
```

6) 如何获取图像的位深度？通常位深度 (bit depth) 与图像存储像素字节位数有关系。

深度为 16 位通常的存储格式为 RGB，前两个通道 red 与 green 各占 5 位，最后一个 blue 通道占 6 位，总计 16 位两个字节，深度为 24 位通常的存储格式为 RGB，分别占 8 位，总计三个字节，深度为 32 位通常包含透明通道，且为 ARGB 格式，每个通道占 1 个字节，总共 32 位，4 个字节。

2.3 基于 `BufferedImageOp` 的图像滤镜演示

通过前面两节的学习，我们已经大致了解 `BufferedImageOp` 接口及其实现类的功能。实践出真知，本节将演示 `BufferedImageOp` 接口中每个实现类的实际使用场景，达到知行合一、学以致用目的，帮助大家解决项目中遇到的实际问题。为了让大家对应用效果有更加深刻的印象，下面会使用 `BufferedImageOp` 的实现类来实现如下几个滤镜特效功能。

- ❑ 黑白滤镜：将彩色图像自动转换为黑白两色图像。
- ❑ 灰度滤镜：将彩色图像自动转换为灰度图像。
- ❑ 模糊滤镜：使图像产生模糊效果。
- ❑ 放缩滤镜：使图像放大或缩小。

1. UI 实现部分

在介绍基于 Swing 的 UI 实现时，关于 Swing UI 部分的编程知识将在下一章中详细剖析与解释，本节的重点放在滤镜实现部分，大致的 UI 布局如图 2-3 所示。

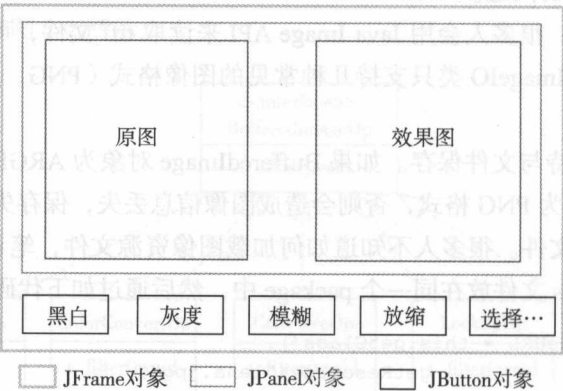


图 2-3 实现界面

2. 滤镜部分的实现

(1) ColorConvertOp 实现灰度功能

ColorConvertOp 主要用于实现各种色彩空间的转换，从而达到转换 BufferedImage 对象类型的目的，也可以在实例化 ColorConvertOp 对象时指定色彩空间。当前支持的色彩空间有五种，实现灰度功能时，只需在实例化 ColorConvertOp 时指定色彩空间为 ColorSpace.CS_GRAY，然后调用它的 filter 方法得到返回图像即可。灰度化的源代码如下：

```
public BufferedImage doColorGray(BufferedImage bi)
{
    ColorConvertOp filterObj = new ColorConvertOp(
        ColorSpace.getInstance(ColorSpace.CS_GRAY), null);
    return filterObj.filter(bi, null);
}
```

(2) LookupOp 实现黑白功能

LookupOp 在实例化时需要传入 LookupTable 实例，当前 LookupTable 接口的两个实现类分别为 ByteLookupTable 与 ShortLookupTable。类关系图 2-4 可以很好地说明它们之间的关系。

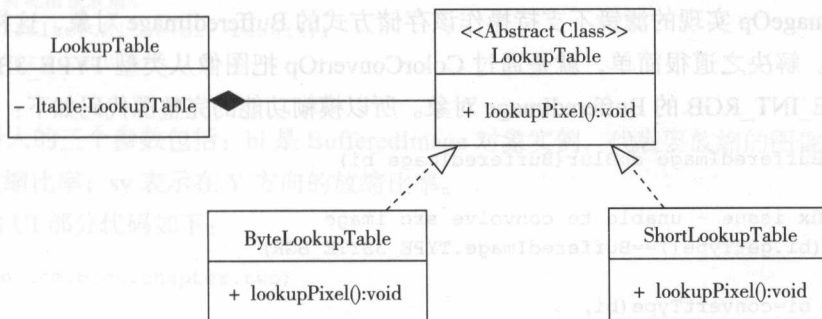


图 2-4 LookupTable 接口与实现类之间关系

运用 LookupOp 实现彩色图像变成黑白单色图像的功能时，首先要将图像灰度化，然后针对灰度图像在 LookupTable 中根据像素值进行索引查找，以便设置新的像素值，从而得到黑白单色图像，代码如下：

```
public BufferedImage doBinaryImage(BufferedImage bi)
{
    bi = doColorGray(bi);
    byte[] threshold = new byte[256];
    for (int i = 0; i < 256; i++)
    {
        threshold[i] = (i < 128) ? (byte)0 : (byte)255;
    }
    BufferedImageOp thresholdOp =
        new LookupOp(new ByteLookupTable(0, threshold), null);
    return thresholdOp.filter(bi, null);
}
```

(3) ConvolveOp 实现模糊功能

ConvolveOp 是实现模板卷积功能操作的类，通过简单设置卷积核 / 卷积模板就可以实现图像模糊功能，实现代码如下：

```
float ninth = 1.0f / 9.0f;
float[] blurKernel = {
    ninth, ninth, ninth,
    ninth, ninth, ninth,
    ninth, ninth, ninth
};
BufferedImageOp blurFilter =
    new ConvolveOp(new Kernel(3, 3, blurKernel));
return blurFilter.filter(bi, null);
```

但是当你想对大多数 JPG 格式图片的 BufferedImage 对象实现模糊功能时，很多情况下 Java 会抛出如下错误消息：

```
unable to convolve src image
```

原因在于 JDK 读入 JPG 格式图像时，多数情况下使用了 TYPE_3BYTE_BGR 存储方式，而 BufferedImageOp 实现的滤镜不支持操作该存储方式的 BufferedImage 对象，这样就导致了上面的错误。解决之道很简单，就是通过 ColorConvertOp 把图像从类型 TYPE_3BYTE_BGR 转换为 TYPE_INT_RGB 的 BufferedImage 对象。所以模糊功能的完整源代码如下：

```
public BufferedImage doBlur(BufferedImage bi)
{
    // fix issue - unable to convolve src image
    if (bi.getType() == BufferedImage.TYPE_3BYTE_BGR)
    {
        bi = convertType(bi,
            BufferedImage.TYPE_INT_RGB);
    }

    float ninth = 1.0f / 9.0f;
    float[] blurKernel = {
        ninth, ninth, ninth,
        ninth, ninth, ninth,
        ninth, ninth, ninth
    };
    BufferedImageOp blurFilter =
        new ConvolveOp(new Kernel(3, 3, blurKernel));
    return blurFilter.filter(bi, null);
}
```

convertType 方法的代码如下：

```
ColorConvertOp cco = new ColorConvertOp(null);
BufferedImage dest = new BufferedImage(
    src.getWidth(), src.getHeight(), type);
```

```
cco.filter(src, dest);
return dest;
```

(4) AffineTransformOp 实现图像 zoom in/out 的功能

AffineTransformOp 支持的操作包括图像的错切、旋转、放缩、平移。要实现图像的放缩功能，首先要通过 AffineTransform.getInstance 来获取 Scale 实例，然后作为参数初始化 AffineTransformOp 对象实例，最后调用 filter 方法即可。实现图像放缩功能的代码如下：

```
public BufferedImage doScale(BufferedImage bi,
                             double sx, double sy)
{
    AffineTransformOp atfFilter = new AffineTransformOp(
        AffineTransform.getInstance(sx, sy),
        AffineTransformOp.TYPE_BILINEAR);
    //计算放缩后图像的宽与高
    int nw = (int)(bi.getWidth() * sx);
    int nh = (int)(bi.getHeight() * sy);
    BufferedImage result = new BufferedImage(
        nw, nh, BufferedImage.TYPE_3BYTE_BGR);
    //实现图像放缩
    atfFilter.filter(bi, result);
    return result;
}
```

需要传入的三个参数包括：bi 是 BufferedImage 对象实例，代表要放缩的图像；sx 表示在 X 方向的放缩比率；sy 表示在 Y 方向的放缩比率。

完整的 UI 部分代码如下：

```
package com.book.chapter.two;

import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;

import javax.imageio.ImageIO;
import javax.swing.JButton;
import javax.swing.JFileChooser;
import javax.swing.JFrame;
import javax.swing.JOptionPane;
import javax.swing.JPanel;

public class MyFilterUI extends JFrame
    implements ActionListener {
```

```

/**
 *
 */
private static final long serialVersionUID = 1L;
public static final String GRAY_CMD = "灰度";
public static final String BINARY_CMD = "黑白";
public static final String BLUR_CMD = "模糊";
public static final String ZOOM_CMD = "放缩";
public static final String BROWSER_CMD = "选择...";

private JButton grayBtn;
private JButton binaryBtn;
private JButton blurBtn;
private JButton zoomBtn;
private JButton browserBtn;
private MyFilters filters;

// image
private BufferedImage srcImage;

public MyFilterUI()
{
    this.setTitle("JAVA 2D BufferedImageOp - 滤镜演示");
    grayBtn = new JButton(GRAY_CMD);
    binaryBtn = new JButton(BINARY_CMD);
    blurBtn = new JButton(BLUR_CMD);
    zoomBtn = new JButton(ZOOM_CMD);
    browserBtn = new JButton(BROWSER_CMD);

    // buttons
    JPanel btnPanel = new JPanel();
    btnPanel.setLayout(new
        FlowLayout(FlowLayout.RIGHT));
    btnPanel.add(grayBtn);
    btnPanel.add(binaryBtn);
    btnPanel.add(blurBtn);
    btnPanel.add(zoomBtn);
    btnPanel.add(browserBtn);

    // filters
    filters = new MyFilters();

    getContentPane().setLayout(new BorderLayout());
    getContentPane().add(filters, BorderLayout.CENTER);
    getContentPane().add(btnPanel, BorderLayout.SOUTH);

    // setup listener
    setupActionListener();
}

```

```

private void setupActionListener() {
    grayBtn.addActionListener(this);
    binaryBtn.addActionListener(this);
    blurBtn.addActionListener(this);
    zoomBtn.addActionListener(this);
    browserBtn.addActionListener(this);
}

```

@Override

```

public void actionPerformed(ActionEvent e) {
    if(srcImage == null)
    {
        JOptionPane.showMessageDialog(this,
            "请先选择图像源文件");
        try {
            JFileChooser chooser = new JFileChooser();
            chooser.showOpenDialog(null);
            File f = chooser.getSelectedFile();
            srcImage = ImageIO.read(f);
            filters.setImage(srcImage);
            filters.repaint();
        } catch (IOException el) {
            el.printStackTrace();
        }
        return;
    }
    if(GRAY_CMD.equals(e.getActionCommand()))
    {
        filters.doColorGray(srcImage);
        filters.repaint();
    }
    else if(BINARY_CMD.equals(e.getActionCommand()))
    {
        filters.doBinaryImage(srcImage);
        filters.repaint();
    }
    else if(BLUR_CMD.equals(e.getActionCommand()))
    {
        filters.doBlur(srcImage);
        filters.repaint();
    }
    else if(ZOOM_CMD.equals(e.getActionCommand()))
    {
        filters.doScale(srcImage, 1.5, 1.5);
        filters.repaint();
    }
    else if(BROWSER_CMD.equals(e.getActionCommand()))
    {
        try {
            JFileChooser chooser = new JFileChooser();
            chooser.showOpenDialog(null);

```

```

        File f = chooser.getSelectedFile();
        srcImage = ImageIO.read(f);
        filters.setImage(srcImage);
        filters.repaint();
    } catch (IOException e1) {
        e1.printStackTrace();
    }
}

public static void main(String[] args) {
    MyFilterUI ui = new MyFilterUI();
    ui.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    ui.setPreferredSize(new Dimension(800, 600));
    ui.pack();
    ui.setVisible(true);
}
}

```

这里主要是基于 JFrame 对象实现 UI 部分，通过重载 JPanel 的 `paintComponent()` 方法来显示原图与处理后的效果图。按钮动作响应通过监听 `ActionListener` 来实现，处理完以后通过调用 `repaint()` 方法来实现 UI 刷新。

2.4 小结

本章重点介绍了 Java 2D 中关于图像方面的操作接口类 `BufferedImageOp`，通过其实现类可以很方便地实现图像的色彩空间转换，自定义颜色查找表，卷积功能（包括边缘提取、线性模糊、高斯模糊），图像的放大与缩小、错切变化、平移变换、旋转变换等。最后本章通过编码实现了几种简单而且常见的图像处理功能，帮助读者加深对 `BufferedImageOp` 接口的认识。如果你还想更加深入地了解 `BufferedImageOp` 接口实现类的使用，请参照 JDK 官方文档说明，同时建议你多多编程实践，只有加深认知才能更好地掌握与运用 `BufferedImageOp` 实现类的功能。

基本 Swing UI 组件与图像显示

上一章介绍了 `BufferedImageOp` 的一些重要知识，实现了几个常见的图像特效，本章介绍如何通过 Swing UI 组件显示与刷新图像。首先会介绍 Java Swing 的顶层组件 `JFrame`，然后介绍 Swing 中最重要和使用频率最高的组件 `JPanel`，教会读者重写 `JComponent` 中的 `paintComponent()` 方法来实现图像的显示，最后会介绍 Swing 组件 `JButton` 捕获与监听用户行为时最重要的 `ActionListener` 接口的使用，以及在 Swing 事件派遣线程中刷新显示等的技巧，希望可指导读者在后续的图像处理实践中，通过 Swing UI 来实现自己的 UI 测试类。本书不是一本专门介绍 Java Swing 编程的图书，因此要求读者对 Java Swing 常见组件有基本认识，对 Swing 事件监听与处理有基本的知识。

本章最主要的目的是实现一个 Java Swing UI，即一个测试框架，来测试第 4 章到第 13 章中所有继承自 `AbstractBufferedImageOp` 抽象类的源代码，帮助读者更好地理解所学到的关于图像处理的知识与内容。

3.1 JPanel 组件与 BufferedImage 对象的显示

刚接触 Swing 编程的读者可能对 `JPanel` 的了解并不多，常常不清楚如何在 `JPanel` 中显示图像，而网上的很多教程又是通过 `JLabel` 来作为 `BufferedImage` 实例显示组件的，这其实不是一种很好的方法，不值得推荐。在 `JPanel` 中显示 `BufferedImage` 对象实例时，值得推荐的做法应该是通过重载 `paintComponent()` 方法来实现图像的显示与及时刷新。这种方法的大致实现可以分为以下几步。

1) 重载 `JPanel` 中的 `paintComponent()` 方法。

2) 获取 Graphics2D 图形引擎绘制对象, 使用 drawImage 方法绘制图像, 代码如下:

```
protected void paintComponent(Graphics g) {
    Graphics2D g2d = (Graphics2D) g;
    g2d.clearRect(0, 0, this.getWidth(),
                  this.getHeight());
    if(sourceImage != null)
    {
        g2d.drawImage(sourceImage, 0, 0,
                      sourceImage.getWidth(),
                      sourceImage.getHeight(), null);
        if(destImage != null)
        {
            g2d.drawImage(destImage,
                          sourceImage.getWidth() + 10 ,
                          0, destImage.getWidth(),
                          destImage.getHeight(), null);
        }
    }
}
```

3) 使用 repaint() 方法及时绘制更新。

以上简单的三步即可实现 BufferedImage 对象实例在 JPanel 的现实与刷新。

根据上述方法实现了一个完整的可以显示与刷新 BufferedImage 对象实例的 ImagePanel 类, 代码如下:

```
package com.book.chapter.three;

import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.image.BufferedImage;

import javax.swing.JPanel;

public class ImagePanel extends JPanel {

    private static final long serialVersionUID = 1L;
    private BufferedImage sourceImage;
    private BufferedImage destImage;
    public ImagePanel()
    {
        // do nothing
    }
    @Override
    protected void paintComponent(Graphics g) {
        Graphics2D g2d = (Graphics2D) g;
        g2d.clearRect(0, 0, this.getWidth(),
                      this.getHeight());
        if(sourceImage != null)
        {
```

```

g2d.drawImage(sourceImage, 0, 0,
sourceImage.getWidth(),
sourceImage.getHeight(), null);
if(destImage != null)
{
    g2d.drawImage(destImage,
    sourceImage.getWidth() + 10 ,
    0, destImage.getWidth(),
    destImage.getHeight(), null);
}
}
public BufferedImage getSourceImage() {
    return sourceImage;
}
public void setSourceImage(BufferedImage sourceImage) {
    this.sourceImage = sourceImage;
}
public BufferedImage getDestImage() {
    return destImage;
}
public void setDestImage(BufferedImage destImage) {
    this.destImage = destImage;
}
}

```

3.2 JFrame 组件与 Main UI 实现

要想真正把读入图像的 `BufferedImage` 对象实例显示到 UI 上为眼睛所见, 还需要使用 `JFrame` 组件, 把 `JPanel` 组件实例通过 `add()` 方法加到 `JFrame` 的内容面板上。在 Java Swing 中只有 `JFrame`、`JDialog` 与 `JApplet` 属于顶层容器, 其他组件最终必须依附于顶层容器才能够正确显示, 使用 `JFrame` 来显示 `JPanel` 与 `BufferedImage` 对象实例大致可以通过如下几步实现。

- 1) 在 `JPanel` 中通过重载 `JComponent` 的 `paintComponent()` 方法绘制 `BufferedImage` 实例。
- 2) 获取 `JFrame` 的内容面板, 这里使用的布局管理器为 `BorderLayout`, 然后把 `JPanel` 实例添加到 `JFrame` 的内容面板中, 代码如下:

```

// JPanel 实例对象添加到 JFrame 的内容面板上
imagePanel = new ImagePanel();
getContentPane().setLayout(new BorderLayout());
getContentPane().add(imagePanel, BorderLayout.CENTER);

```

- 3) 通过 `JFrame` 的 `setVisible()` 方法来实现 `JFrame` 的显示, 通过 `setPreferredSize()` 方法来控制 `JFrame` 组件的大小。代码如下:

```

setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

```

```
setPreferredSize(new Dimension(800, 600));
pack();
setVisible(true);
```

3.3 JFileChooser 文件选择框的使用

介绍 Swing 中的 JFileChooser 文件选择框是因为我们经常会用它实现选择本地图片文件，然后加载到 JPanel 组件中显示，JFileChooser 组件类的使用方法极其简单，只要简单的三行代码就可以提供相应的文件选择对话框，代码如下：

```
JFileChooser chooser = new JFileChooser();
chooser.showOpenDialog(null);
File f = chooser.getSelectedFile();
```

如果想在文件选择对话框中只看到指定类型的文件，则可以通过 setFileFilter() 来实现。一个最简单的支持选择图像格式文件的 FileFilter 示例的代码如下：

```
FileNameExtensionFilter filter =
    new FileNameExtensionFilter(
        "JPG & PNG Images", "jpg", "png");
chooser.setFileFilter(filter);
```

这样就可以实现文件类型的过滤了，在打开时只会看到 JPG 与 PNG 格式的图片文件，其他类型文件则会被自动过滤。从上述代码也可以看到，在 Java Swing 中使用文件选择框是非常简单与方便的。

3.4 基本 JButton 事件响应

在学习 JButton 事件响应的知识之前，首先来看一下 Swing 中如何实现对用户事件的监听与处理，认识一下 Swing 中事件响应最重要的线程——事件分派线程。

在 Swing 中有一个特殊的线程被称为 Swing 事件分配线程，如果对 UI 组件的操作不在 Swing 事件分派线程中，Swing 将抛出异常。检测当前线程是否为事件分派线程可以通过 Swing 本身提供的一个简单方法 SwingUtilities.isEventDispatchThread() 来完成。对 Swing UI 组件的刷新、重绘等必须都在事件分派线程中完成，这是因为 Swing 组件本身的设计不是线程安全的，所以通过一个特殊的线程——事件分派线程来实现对所有组件的更新与重绘，这样就保证了 Swing 组件操作的线程安全性。

JButton 组件是 Java Swing 中实现用户交互最常用的组件，当用户单击对应的 JButton 组件时，Swing 通过监听组件添加 ActionListener 对象实例，来实现对 JButton 组件单击事件的监听与响应处理，其响应处理则通过实现 ActionListener 接口的 actionPerformed() 方法来完成。很多时候，在单击 JButton 按钮以后，虽然有很多事情要做，但还是希望 UI 可以继续响应用户操作，这时可使用 SwingWork 来完成用户的操作并刷新 UI 显示。

在对 Swing 事件响应机制有初步了解以后,下面看一下在正式的项目编程中 Swing 如何实现对 JButton 事件监听与响应。其实现过程大致可以分为两步完成。

- 1) 实现 ActionListener 接口,最常见的是由定义 JButton 组件的类来实现 ActionListener 接口。
- 2) 根据 ActionEvent.getActionCommand() 得到的文本常量响应相应的用户操作。

3.5 一个完整的 Swing UI Demo

本节将根据前面四节所讲的 Swing UI 组件应用知识,实现一个真正的 Swing UI 演示,以更加贴近实际编程的方式来说明 Swing 中组件的应用知识。首先来介绍一下要实现的功能:

- ❑ 通过文件对话框选择图像文件,刷新 JFrame 中的内容面板实现图像显示。
- ❑ 通过单击 [处理] 按钮实现对图像的必要时处理,然后刷新显示图像。

大致的 UI 组件布局如图 3-1 所示。

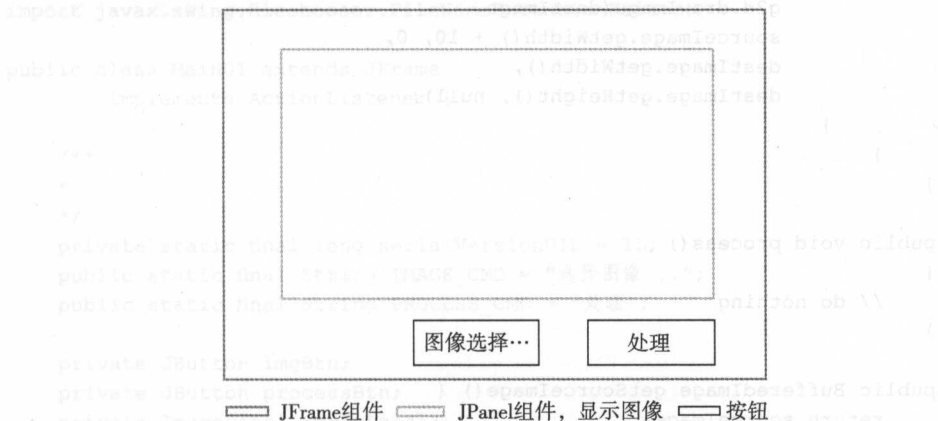


图 3-1 UI 布局结构

其中支持 BufferedImage 对象显示的自定义 JPanel 类的实现如下:

```
package com.book.chapter.three;

import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.image.BufferedImage;

import javax.swing.JPanel;

public class ImagePanel extends JPanel {
    /**
     *
     */
    private static final long serialVersionUID = 1L;
    private BufferedImage sourceImage;
```



```

private BufferedImage destImage;

public ImagePanel() {
}

@Override
protected void paintComponent(Graphics g) {
    Graphics2D g2d = (Graphics2D) g;
    g2d.clearRect(0, 0,
        this.getWidth(),
        this.getHeight());
    if (sourceImage != null) {
        g2d.drawImage(sourceImage, 0, 0,
            sourceImage.getWidth(),
            sourceImage.getHeight(), null);
        if (destImage != null) {
            g2d.drawImage(destImage,
                sourceImage.getWidth() + 10, 0,
                destImage.getWidth(),
                destImage.getHeight(), null);
        }
    }
}

public void process() {
    // do nothing
}

public BufferedImage getSourceImage() {
    return sourceImage;
}

public void setSourceImage(BufferedImage sourceImage) {
    this.sourceImage = sourceImage;
}

public BufferedImage getDestImage() {
    return destImage;
}

public void setDestImage(BufferedImage destImage) {
    this.destImage = destImage;
}
}

```

SwingUI 界面实现与 JButton 按钮监听处理的类的代码如下：

```
package com.book.chapter.three;
```

```

import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;

import javax.imageio.ImageIO;
import javax.swing.JButton;
import javax.swing.JFileChooser;
import javax.swing.JFrame;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.SwingUtilities;
import javax.swing.filechooser.FileNameExtensionFilter;

public class MainUI extends JFrame
    implements ActionListener {

    /**
     *
     */
    private static final long serialVersionUID = 1L;
    public static final String IMAGE_CMD = "选择图像...";
    public static final String PROCESS_CMD = "处理";

    private JButton imgBtn;
    private JButton processBtn;
    private ImagePanel imagePanel;

    // image
    private BufferedImage srcImage;

    public MainUI()
    {
        setTitle("JFrame UI - 演示");
        imgBtn = new JButton(IMAGE_CMD);
        processBtn = new JButton(PROCESS_CMD);

        // buttons
        JPanel btnPanel = new JPanel();
        btnPanel.setLayout(new
            FlowLayout(FlowLayout.RIGHT));
        btnPanel.add(imgBtn);
        btnPanel.add(processBtn);

        // filters
        imagePanel = new ImagePanel();

```

```

getContentPane().setLayout(new BorderLayout());
getContentPane().add(imagePanel,
    BorderLayout.CENTER);
getContentPane().add(btnPanel, BorderLayout.SOUTH);

// setup listener
setupActionListener();

private void setupActionListener() {
    imgBtn.addActionListener(this);
    processBtn.addActionListener(this);
}

@Override
public void actionPerformed(ActionEvent e) {
    if(SwingUtilities.isEventDispatchThread())
    {
        System.out.println("Event Dispath Thread!!");
    }

    if(srcImage == null)
    {
        JOptionPane.showMessageDialog(this,
            "请先选择图像源文件");
        try {
            JFileChooser chooser =
                new JFileChooser();
            setFileTypeFilter(chooser);
            chooser.showOpenDialog(null);
            File f = chooser.getSelectedFile();
            if(f != null)
            {
                srcImage = ImageIO.read(f);
                imagePanel.setSourceImage(
                    srcImage);
                imagePanel.repaint();
            }
        } catch (IOException e1) {
            e1.printStackTrace();
        }
        return;
    }
    if (IMAGE_CMD.equals(e.getActionCommand())) {
        try {
            JFileChooser chooser =
                new JFileChooser();

```

```

        setFileTypeFilter(chooser);
        chooser.showOpenDialog(null);
        File f = chooser.getSelectedFile();
        if(f != null)
        {
            srcImage = ImageIO.read(f);
            imagePanel.setSourceImage(
                srcImage);
            imagePanel.repaint();
        }
    } catch (IOException e1) {
        e1.printStackTrace();
    }
    imagePanel.repaint();
}
else if (PROCESS_CMD.equals(e.getActionCommand()))
{
    imagePanel.process();
    imagePanel.repaint();
}
}

public void setFileTypeFilter(JFileChooser chooser)
{
    FileNameExtensionFilter filter =
        new FileNameExtensionFilter(
            "JPG & PNG Images", "jpg", "png");
    chooser.setFileFilter(filter);
}

public void openView()
{

```

```

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setPreferredSize(new Dimension(800, 600));
        pack();
        setVisible(true);
    }
}

```

```

public static void main(String[] args) {

```

```

    MainUI ui = new MainUI();
    ui.openView();
}

```

3.6 小结

本章一步一步地剖析如何了构建一个 Swing UI 程序，介绍了 JPanel、JButton、JFileChooser 等组件的用法，最后通过 JFrame 组件组合成为用户交互界面，实现了对图像文件的

显示与操作，以及 UI 响应用户的操作与刷新。这也是本书后面多数章节中要用到的测试 UI，所以学习与掌握本章知识，将为后面图像处理的代码提供一个 UI 现实与效果演示界面，帮助读者加深对知识的理解。前面三章已经介绍了 Java 图像处理 API 基础知识与 Swing 的基础知识，这为后面学习图像处理做了很好的铺垫，特别是其中的像素操作方法，这将在后面的编程中一直使用。

本章的主要目的不是介绍 Swing 编程知识，如果读者对 Swing 编程感兴趣的话，可以阅读 JDK 官方关于 Swing 编程的技术文档。

本章进一步详细讲解如何构建一个 Swing UI 界面，介绍了 JPanel、JButton、JFile-Chooser 等组件的用法，最后通过 JFrame 组件组合成对图像文件交互界面，实现了对图像文件的



第4章 Chapter 4

图像属性

前三章讲的是 Java 图形编程与 Swing 的基础知识，本章开始我们将注意力放到图像处理本身。直观地讲，图像属性可以指图像的大小即字节数、图像的宽与高、位图的深度、图像的压缩与存储格式等，进一步提取的话可以包括图像的色彩空间、图像的直方图、亮度、像素值、透明通道等。图像的这些属性是它最重要的基础数据，在了解图像这些属性的基础上，通常我们可以通过改变图像的这些属性值来实现对图像处理，从而达到想要的各种结果。本章的内容就是先介绍这些属性，然后通过改变这些属性实现对图像一些简单处理，从而达到不同的视觉效果，同时也让读者初步认知图像处理的基本步骤与流程。



注意 如果没有特别说明，本章及其后续所有章节默认图像的色彩空间都是 RGB 空间。

4.1 失去的时光与回忆——老照片特效

有一首很经典的英文歌曲叫《昔日重来》，然而随着岁月的流逝，我们慢慢知道了昔日不会再来，那些微微泛黄的照片成了昔日最好时光的见证。也许是为了满足人们那难以割舍的怀旧情结，人们有时会想让图像看上去微微泛黄，像是一张年代久远的老照片。其实，这可以通过对图像像素值进行调整来实现，英文中称这种特效为 Sepia Tone Effect，在绝大多数的图像处理应用软件中，它已经属于标配功能。

它的实现大致可以分为以下三步完成。

1) 首先对图像的每个像素点重新计算 RGB 值，代码如下。

```
int fr= (int)(((double)tr * 0.393) + ((double)tg * 0.769)
        + ((double)tb * 0.189));
int fg = (int)(((double)tr * 0.349) + ((double)tg * 0.686)
        + ((double)tb * 0.168));
int fb= (int)(((double)tr * 0.272) + ((double)tg * 0.534)
        + ((double)tb * 0.131));
```

2) 获取混合的权重系数, 通过随机方法获取, 取值范围为 $[0 \sim 1]$, 代码如下。

```
private double noise() {
    return Math.random()*0.5 + 0.5;
}
```

3) 根据权重系数, 将该像素点的原值与第一步中得到新值混合, 从而得到该像素最终值, 代码如下。

```
private double colorBlend(double scale, double dest, double src)
{
    return (scale * dest + (1.0 - scale) * src);
}
```

其中 `scale` 表示权重、`dest` 表示新像素值、`src` 表示原来像素值。

当然本例还涉及其他两个编程问题, 一个是如何从 `BufferedImage` 对象中读取像素数据, 以及如何将处理后的像素数据写回到 `BufferedImage` 对象中。为了让程序简单易懂, 这里通过前面学习的知识实现了一个抽象类 `AbstractBufferedImageOp`, 然后会把对像素数据的读写都放在抽象类中完成, 这样就实现了代码的重用, 也为处理像素提供了更加方便的公共方法。关于如何读取与写入像素数据在前面的章节中已经详细阐述, 在此不再重复。`AbstractBufferedImageOp` 类的实现代码可以参见下载的源文件。

完全版实现老照片效果的代码如下:

```
package com.gloomyfish.filter.study;

import java.awt.image.BufferedImage;

public class SepiaToneFilter extends AbstractBufferedImageOp {

    @Override
    public BufferedImage filter(BufferedImage src, BufferedImage dest) {
        int width = src.getWidth();
        int height = src.getHeight();
        if (dest == null)
            dest = createCompatibleDestImage(src, null);

        int[] inPixels = new int[width*height];
        int[] outPixels = new int[width*height];
        getRGB(src, 0, 0, width, height, inPixels);
        int index = 0;
        for(int row=0; row<height; row++) {
```

```

int ta = 0, tr = 0, tg = 0, tb = 0;
for(int col=0; col<width; col++) {
    index = row * width + col;
    ta = (inPixels[index] >> 24) & 0xff;
    tr = (inPixels[index] >> 16) & 0xff;
    tg = (inPixels[index] >> 8) & 0xff;
    tb = inPixels[index] & 0xff;

    int fr = (int)colorBlend(noise(), (tr * 0.393) + (tg * 0.769) +
        (tb * 0.189), tr);
    int fg = (int)colorBlend(noise(), (tr * 0.349) + (tg * 0.686) +
        (tb * 0.168), tg);
    int fb = (int)colorBlend(noise(), (tr * 0.272) + (tg * 0.534) +
        (tb * 0.131), tb);

    outPixels[index] = (ta << 24) | (clamp(fr) << 16) | (clamp(fg) <<
        8) | clamp(fb);
}
setRGB( dest, 0, 0, width, height, outPixels );
return dest;
}

private double noise() {
    return Math.random()*0.5 + 0.5;
}

private double colorBlend(double scale, double dest, double src) {
    return (scale * dest + (1.0 - scale) * src);
}

public static int clamp(int c)
{
    return c > 255 ? 255 : ( c < 0 ) ? 0 : c;
}

public String toString()
{
    return "Sepia Tone Effect - Effect from Photoshop App";
}
}

```

如果你还记得本书第3章的内容，应该知道只要在 ImagePanel 类的 process() 方法中加上如下的代码就可以得到上述滤镜的运行效果：

```

SepiaToneFilter filter = new SepiaToneFilter();
destImage = filter.filter(sourceImage, null);

```

运行第3章中已经完成的测试 UI-MainUI.java，然后选择图像，单击【处理】按钮以后，

就可以看到程序运行的结果了。



注意 如果没有特别说明，本章及其后续所有章节默认对继承自 `AbstractBuffered-ImageOp` 实现类的代码中与使用的测试方式都与本节中 `SepiaToneFilter` 的测试方式完全一致。

4.2 图像属性

上一节只是提供了对图像处理的直观感受，我们还需把图像属性一一呈现出来，为后面更加深入的学习打下基础，下面内容将由浅入深地介绍图像的这些属性，并讲解如何获取它们。

1. 图像的宽与高

图像的宽度与高度是图像的基本属性信息，在一般的图像处理中，第一步都是获取图像的宽度与高度，Java 语言通过 `BufferedImage` 对象可以很容易地获取图像宽度与高度。代码如下：

```
int width = image.getWidth();  
int height = image.getHeight();
```

2. 图像的像素值

像素的像素值是图像中最重要的属性之一，如何正确提取像素值，对后面的处理很关键，提取像素值在各种不同的编程语言中都有对应的 API 接口，在 Java 中提取图像像素数据已经在前面做了非常详细的介绍与代码演示，在此不再赘述。

3. 图像类型

图像文件最常见的类型是 JPG、GIF、PNG 格式，其中 PNG 格式的最初设计目的是为了网络图形传输方便，该格式的图像为 24 位与 32 位的带透明通道的图像，只支持 RGB 空间。另外一种很常见的图像格式为 Windows 位图格式 BMP 类型。无论是哪种图形格式，都有一定的读写存储固定格式，如果不知道如何读取一种图像类型，最好的方法是查阅官方文档，对照格式说明完成代码即可实现对图像的读写。JPG 图像类型的解码与编码涉及多种算法，包括行程编码、霍夫曼编码、离散余弦变换等，JPG2000 以后的版本还会用小波变换算法作为压缩算法。其他的图像格式类型还包括 tiff、RAW 等。

4. 色彩空间

色彩空间对图像的显示效果与处理有很大的影响，有些图像处理手段只有在特定的色彩空间才会得到很好的效果，所以色彩空间是图像非常重要的一个属性，常见的色彩空间有 RGB、HSL/HSV、YCrCb 等，下面就分别介绍这些色彩空间。

RGB 色彩空间，通过三种单色红 (R)、绿 (G)、蓝 (B) 组合而成，取值范围为 $[0 \sim 255]$ ，其

色彩空间支持超过 1000 万种的不同颜色，如果 RGB(0,0,0) 都为 0 则表示黑色，RGB(255,255,255) 都为 255 表示白色。大多数图像默认的颜色空间均为 RGB 色彩空间。

HSL/HSV 色彩空间，分别包含 Hue 通道、饱和度 (Saturation) 通道、亮度 (Lightness) 通道，常见的调整图像的亮度与饱和度一般都是先将图像像素值从 RGB 空间转换到 HSL 空间，待调整完毕以后再重新转换回到 RGB 色彩空间。将像素值从 RGB 色彩空间转换到 HSL 色彩空间的代码如下：

```
// convert to HSL space
min = tr;
if (tg < min)
    min = tg;
if (tb < min)
    min = tb;
max = tr;
f1 = 0.0;
f2 = tg - tb;
if (tg > max) {
    max = tg;
    f1 = 120.0;
    f2 = tb - tr;
}
if (tb > max) {
    max = tb;
    f1 = 240.0;
    f2 = tr - tg;
}
dif = max - min;
sum = max + min;
l = 0.5 * sum;
if (dif == 0) {
    h = 0.0;
    s = 0.0;
}
else if (l < 127.5) {
    s = 255.0 * dif / sum;
}
else {
    s = 255.0 * dif / (510.0 - sum);
}
h = (f1 + 60.0 * f2 / dif);
if (h < 0.0) {
    h += 360.0;
}
if (h >= 360.0) {
    h -= 360.0;
}
```



```

    h1 += 360.0;
}
if (h1 < 60.0) {
    tb = (int)(v1 + v3 * h1 * c1o60);
}
else if (h1 < 180.0) {
    tb = (int)v2;
}
else if (h1 < 240.0) {
    tb = (int)(v1 + v3 * (4 - h1 * c1o60));
}
else {
    tb = (int)v1;
}
}
}

```

YCrCb 色彩空间的三个通道中 Y 的取值范围为 [16 ~ 235], Cr 与 Cb 的取值范围均为 [16 ~ 240]。这种颜色空间的设计初衷是为了压缩 RGB 值, 可以使用更少的空间来携带相同的颜色值。根据标准不同, 从 RGB 到 YCrCb 的转换公式会稍有不同, 最常见的像素值 RGB 色彩空间变换到 YCrCb 色彩空间的代码如下:

```

int y = (int)(tr * 0.299 +
    tg * 0.587 +
    tb * 0.114);
int Cr = tr - y;
int Cb = tb - y;

```

4.3 图像的亮度、对比度和饱和度

图像的亮度、对比度与饱和度是图像处理与美化中经常需要编辑的三个属性, 也是很多图像处理软件的基本功能之一, 本节首先从原理上认识一下如何调整这三个属性, 在后续章节中将会有专门的编程实践与应用实例。

1. 亮度 (Brightness/lightness)

图像的亮度从本质上来说是像素灰度值的强度, 取值在 0 ~ 255 之间, 0 表示黑色, 255 表示白色亮度值最大, 对 RGB 图像来说, RGB (0, 0, 0) 表示黑色, RGB (255, 255, 255) 表示白色, 调整图像的亮度就是对图像的像素值在 RGB 每个分量上计算平均值, 然后用平均值乘以亮度系数, 当系数值为 1 时表示图像亮度不变, 当亮度值小于 1 时, 图像将比原图暗一点, 当亮度值大于 1 时, 则图像会比原图亮一点。这种调整亮度的方法比较直观明了, 编程难度也比较小, 容易实现。另外一种更常见的是把图像从 RGB 色彩空间转换到 HSL 色彩空间, 这样亮度就对应 L 分量, 可以直接调整, 调整以后再转换到 RGB 色彩空间即可。两种方法各有各的优点与缺点, 方法一简单直接, 计算量小, 但是效果可能不是特别好, 方法二精确度高, 效果好。

2. 对比度 (Contrast)

从像素值上来说，提升图像对比度就是让像素值之间的差异更加显著，从而更加突出图像的细节与特征；降低图像的对比度则是让像素值之间的差异减小，从而更多地隐藏图像细节。调整图像对比度的算法有很多，最直观与常见的算法步骤如下：

- 1) 计算图像像素的 RGB 每个分量的平均值。
- 2) 对每个待调整的像素减去平均值。
- 3) 对步骤 2 的结果乘以对比度调整系数，1 表示保存不变，大于 1 表示提高对比度，小于 1 表示减低对比度。
- 4) 对步骤 3 的结果加上 RGB 分量的平均值，即为调整以后的像素值。
- 5) 归一化处理，确保每个像素值落在 0 ~ 255 之间。

3. 饱和度 (Saturation)

图像饱和度主要针对色彩空间是 HSV/HSL 的 S 分量，即饱和度分量，对 RGB 图像，要想实现对图像饱和度的调整，首先要把像素值从 RGB 色彩空间转换到 HSV/HSL 色彩空间，待调整完毕以后再转换回到 RGB 色彩空间。提高图像的饱和度后，图像会非常明亮；若降低图像的饱和度，图像则看上去很暗。

4.4 图像饱和度调整

从上一节的内容不难看出，对图像饱和度的调整可以大致分如下几步进行：

- ❑ 读取图像像素，将像素值从 RGB 色彩空间转到 HSL 色彩空间。
- ❑ 在 HSL 色彩空间调整 S 分量，即饱和度分量。
- ❑ 将第二步处理完的像素值从 HSL 色彩空间转换到 RGB 色彩空间。

下面来看看编程关键技巧。

1. HSL 分量取值范围与 S 分量值处理

对于 HSL 的三个分量取值范围，H 的取值范围为 [0 ~ 360]，其余两个分量的取值范围均为 [0 ~ 255]，在实际编程中对于超过范围的 S 分量值，可以如下处理：

```
s = s + sat;
if ( s < 0.0 ) {
    s = 0.0;
}
if ( s > 255.0 ) {
    s = 255.0;
}
```

其中 s 表示饱和度值，sat 表示调整值，其计算公式为 $\text{sat} = 127 * \text{ratio}$ ，其中 ratio 的取值范围为 [-1, 1]，这样就最终得到处理以后的 s 值，即调整以后的饱和度值。

2. HSL 与 RGB 相互转化

像素值在 HSL 与 RGB 色彩空间相互转换的代码前面一节已给出，可以直接拿过来用，这里需要特别说明一下，HSL 到 RGB 色彩空间的转换非常有用，很多图像处理算法都是在 HSL/Hue 色彩空间提取特征与图像分割的。

3. 饱和度调整系数 ratio

其取值范围为 $[-1, 1]$ ，这里给出了一个默认值，即为 0.25，通常情况下，稍微调整可以使人更容易接受。可以在初始化调整饱和度类的时候传入一个你想要参数。

饱和度调整程序继承了 `BufferedImageOP` 接口，看上去程序更加通用，你几乎不用做任何修改就可以在应用程序使用该类。完整的源代码如下：

```
package com.book.chapter.four;

import java.awt.image.BufferedImage;

public class SaturationFilter extends AbstractBufferedImageOp {
    public final static double clo60 = 1.0 / 60.0;
    public final static double clo255 = 1.0 / 255.0;
    private double ratio = 0.25;

    public SaturationFilter(double ratio) {
        this.ratio = ratio;
    }

    public double getRatio() {
        return ratio;
    }

    public void setRatio(double ratio) {
        this.ratio = ratio;
    }

    @Override
    public BufferedImage filter(BufferedImage src,
                               BufferedImage dest) {
        int width = src.getWidth();
        int height = src.getHeight();
        double sat = 127.0d * ratio;
        if (dest == null)
            dest = createCompatibleDestImage(src, null);

        int[] inPixels = new int[width*height];
        int[] outPixels = new int[width*height];
        getRGB(src, 0, 0, width, height, inPixels);
        double min, max, dif, sum;
        double f1, f2;
        int index = 0;
        double h, s, l;
```

2. 对比度

```
double v1, v2, v3, h1;
for(int row=0; row<height; row++) {
    int ta = 0, tr = 0, tg = 0, tb = 0;
```

```
    for(int col=0; col<width; col++) {
        index = row * width + col;
        ta = (inPixels[index] >> 24) & 0xff;
        tr = (inPixels[index] >> 16) & 0xff;
        tg = (inPixels[index] >> 8) & 0xff;
        tb = inPixels[index] & 0xff;
```

```
// convert to HSL space
```

```
min = tr;
```

```
if (tg < min)
```

```
    min = tg;
```

```
if (tb < min)
```

```
    min = tb;
```

```
max = tr;
```

```
f1 = 0.0;
```

```
f2 = tg - tb;
```

```
if (tg > max) {
```

```
    max = tg;
```

```
    f1 = 120.0;
```

```
    f2 = tb - tr;
```

```
}
```

```
if (tb > max) {
```

```
    max = tb;
```

```
    f1 = 240.0;
```

```
    f2 = tr - tg;
```

```
}
```

```
dif = max - min;
```

```
sum = max + min;
```

```
l = 0.5 * sum;
```

```
if (dif == 0) {
```

```
    h = 0.0;
```

```
    s = 0.0;
```

```
}
```

```
else if(l < 127.5) {
```

```
    s = 255.0 * dif / sum;
```

```
}
```

```
else {
```

```
    s = 255.0 * dif / (510.0 - sum);
```

```
}
```

```
h = (f1 + 60.0 * f2 / dif);
```

```
if (h < 0.0) {
```

```
    h += 360.0;
```

```
}
```

```
if (h >= 360.0) {
```

```
    h -= 360.0;
```

```
}
```

很多图像处理软件 // adjust saturation. HSL Hue 色空间中实现的, 因为在 HSL 色彩空间中 L 分量即表示亮度, 容易理解。通过改变 HSL 中的 L 分量实现亮度调整的大致步骤与调整饱和 s 类似, 唯一不同的是对 L 分量所进行的计算, 其他保持不变, 这里就不再赘述。

下面来看看编程实现:

```

s = s + sat;
if( s < 0.0) {
    s = 0.0;
}
if( s > 255.0) {
    s = 255.0;
}
// conversion back to RGB space here!!
if( s == 0) {
    tr = (int)l;
    tg = (int)l;
    tb = (int)l;
} else {
    if( l < 127.5) {
        v2 = clo255 * l * (255 + s);
    } else {
        v2 = l + s - clo255 * s * l;
    }
    v1 = 2 * l - v2;
    v3 = v2 - v1;
    h1 = h + 120.0;
    if( h1 >= 360.0)
        h1 -= 360.0;

    if( h1 < 60.0) {
        tr = (int)(v1 + v3 * h1 * clo60);
    }
    else if( h1 < 180.0) {
        tr = (int)v2;
    }
    else if( h1 < 240.0) {
        tr = (int)(v1 + v3 * (4 - h1 * clo60));
    }
    else {
        tr = (int)v1;
    }
    h1 = h;
    if( h1 < 60.0) {
        tg = (int)(v1 + v3 * h1 * clo60);
    }
    else if( h1 < 180.0) {
        tg = (int)v2;
    }
    else if( h1 < 240.0) {
        tg = (int)(v1 + v3 * (4 - h1 * clo60));
    }
    else {
        tg = (int)v1;
    }
    h1 = h;
    if( h1 < 60.0) {
        tb = (int)(v1 + v3 * h1 * clo60);
    }
    else if( h1 < 180.0) {
        tb = (int)v2;
    }
    else if( h1 < 240.0) {
        tb = (int)(v1 + v3 * (4 - h1 * clo60));
    }
    else {
        tb = (int)v1;
    }
}

```

完整的代码如下:

```

package com.rooftop.image;
import java.awt.image.BufferedImage;
import java.awt.image.WritableRaster;

public class BrightnessAdjuster {
    private final double clo255 = 255.0;
    private final double clo60 = 60.0;

    public BrightnessAdjuster() {
    }

    public BufferedImage adjust(BrightnessAdjuster b, BufferedImage src) {
        WritableRaster srcRaster = src.getRaster();
        int width = srcRaster.getWidth();
        int height = srcRaster.getHeight();
        int[] data = new int[width * height * 3];
        srcRaster.getData(0, 0, width, height, data);

        for( int i = 0; i < data.length; i += 3) {
            int r = data[i];
            int g = data[i + 1];
            int b = data[i + 2];

            r = b.adjust(r);
            g = b.adjust(g);
            b = b.adjust(b);

            data[i] = r;
            data[i + 1] = g;
            data[i + 2] = b;
        }

        BufferedImage dst = new BufferedImage(width, height, src.getType());
        dst.setRaster(srcRaster);
        return dst;
    }

    private int adjust(int v) {
        if( v < 0)
            return 0;
        if( v > 255)
            return 255;
        return v;
    }
}

```



```

        else {
            tg = (int)v1;
        }
        h1 = h - 120.0;
        if (h1 < 0.0) {
            h1 += 360.0;
        }
        if (h1 < 60.0) {
            tb = (int)(v1 + v3 * h1 * clo60);
        }
        else if (h1 < 180.0) {
            tb = (int)v2;
        }
        else if (h1 < 240.0) {
            tb = (int)(v1 + v3 * (4 - h1 * clo60));
        }
        else {
            tb = (int)v1;
        }
        outPixels[index] = (ta << 24) | (tr << 16) |
            (tg << 8) | tb;
    }
}

setRGB( dest, 0, 0, width, height, outPixels );
return dest;
}
}

```

4.5 图像亮度调整

图像亮度调整在 RGB 色彩空间与 HSL 色彩空间上都可以实现，而且都比较直观易懂。首先来看一下在 RGB 色彩空间进行图像亮度调整的方法步骤：

- 1) 计算像素在 R、G、B 三个分量上的平均值 (means)。
- 2) 在第一步基础上对三个平均值分别乘以对应的亮度系数 brightness，默认值为 1 表示亮度不变，大于 1 表示亮度提高，小于表示亮度降低。
- 3) 对每个像素值在 R、G、B 上的分量，首先减去第一步计算出来的平均值，然后再加上第二步的计算结果。

总结上述三步，调整图像亮度的公式可以简单归纳为：

$$P_{\text{new}} = P_{\text{old}} + (\text{brightness} - 1) * \text{means}$$

其中 P_{new} 表示处理以后的像素， P_{old} 表示处理以前的像素，brightness 表示亮度系数，取值范围为 $[0 \sim 3]$ ，means 表示图像像素的平均值。

很多图像处理软件中的亮度调整都是基于 HSL/Hue 色彩空间完成的,因为在 HSL 色彩空间中 L 分量即表示亮度分量,调整很直观、容易理解,通过改变 HSL 中的 L 分量实现亮度调整的大致步骤与调整饱和度的步骤极其相似,唯一不同的是对 L 分量所进行的计算,其他保持不变,这里就不再赘述。

下面来看看编程关键技巧。

计算图像像素平均值需要将变量定义为 double 类型,图像总的像素数为图像的宽度乘以图像的高度。在实现该类时,同样继承了 AbstractBufferedImageOp 的抽象类,这样就可以重用像素数组读写的代码了。处理以后的像素值可能小于 0 或大于 255,所以对超出 0 ~ 255 这个范围的像素值,如果小于 0 则赋值为 0,大于 255 则赋值为 255,这样就可保证所有的像素值都落在 RGB 的取值范围之内。

完整的代码如下:

```
package com.book.chapter.four;

import java.awt.image.BufferedImage;

public class BrightFilter extends AbstractBufferedImageOp {
    private float brightness;

    public BrightFilter() {
        this(1.2f);
    }

    public BrightFilter(float bright) {
        this.brightness = bright;
    }

    public float getBrightness() {
        return brightness;
    }

    public void setBrightness(float brightness) {
        this.brightness = brightness;
    }

    @Override
    public BufferedImage filter(BufferedImage src,
                               BufferedImage dest) {
        int width = src.getWidth();
        int height = src.getHeight();

        if (dest == null)
            dest = createCompatibleDestImage(src, null);

        int[] inPixels = new int[width * height];
        int[] outPixels = new int[width * height];
```

```

src.getRGB(0, 0, width, height, inPixels, 0, width);

// calculate RED, GREEN, BLUE means of pixel
int index = 0;
int[] rgbmeans = new int[3];
double redSum = 0, greenSum = 0, blueSum = 0;
double total = height * width;
for (int row = 0; row < height; row++) {
    int ta = 0, tr = 0, tg = 0, tb = 0;
    for (int col = 0; col < width; col++) {
        index = row * width + col;
        ta = (inPixels[index] >> 24) & 0xff;
        tr = (inPixels[index] >> 16) & 0xff;
        tg = (inPixels[index] >> 8) & 0xff;
        tb = inPixels[index] & 0xff;
        redSum += tr;
        greenSum += tg;
        blueSum += tb;
    }
}

// get means
rgbmeans[0] = (int) (redSum / total);
rgbmeans[1] = (int) (greenSum / total);
rgbmeans[2] = (int) (blueSum / total);

// adjust brightness algorithm, here
for (int row = 0; row < height; row++) {
    int ta = 0, tr = 0, tg = 0, tb = 0;
    for (int col = 0; col < width; col++) {
        index = row * width + col;
        ta = (inPixels[index] >> 24) & 0xff;
        tr = (inPixels[index] >> 16) & 0xff;
        tg = (inPixels[index] >> 8) & 0xff;
        tb = inPixels[index] & 0xff;

        // remove means
        tr -= rgbmeans[0];
        tg -= rgbmeans[1];
        tb -= rgbmeans[2];

        // adjust brightness
        tr += (int) (rgbmeans[0] * getBrightness());
        tg += (int) (rgbmeans[1] * getBrightness());
        tb += (int) (rgbmeans[2] * getBrightness());
        outPixels[index] = (ta << 24) | (clamp(tr) << 16)
            | (clamp(tg) << 8) | clamp(tb);
    }
}

setRGB(dest, 0, 0, width, height, outPixels);
return dest;

```

```

    }

    public int clamp(int value) {
        return value > 255 ? 255 :
            (value < 0 ? 0 : value);
    }
}

```


测试该功能只需要调用第3章提到的 ImagePanel 类的 process 方法即可。

4.6 图像对比度调整

在具体讲述图像对比度调整之前，首先介绍一下图像对比度的概念及其意义。所谓图像对比度，就是对图像颜色与亮度差异感知，对比度越大，图像的对象与周围差异性也就越大，反之亦然。高的对比图将显示出更多的图像细节差异，低的对比度将会缩小图像像素值在颜色与亮度上的差异。从上面的解释不难得出，要调整图像对比度，还是在于如何调整图像的各个像素值而不改变图像原有的信息。

受上一节内容的启发，对图像像素值求取平均值以后，各个像素减去平均值得到的就是各像素之间的差值，将这些差值乘以一定的对比度系数，然后再加上平均值即可得到调整以后的像素值。但是上节在计算像素平均值时，很多时候会由于图像分辨率很高、像素数据较多而造成计算过度，所以更简洁地调整图像对比度的大致方法如下（前提为对比度系数用户输入范围是 $[-100 \sim 100]$ ）：

- 1) 读取每个 RGB 像素值 Prgb, $Crgb = Prgb/255$ ，使其值范围为 $[0 \sim 1]$ 。
- 2) 基于第一步计算结果 $((Crgb - 0.5) * contrast + 0.5) * 255$ 。
- 3) 第二步中得到的结果就是处理以后的像素值。

 **注意** 在第三步中必须检查处理以后的结果，如果值大于 255，则 255 为处理后的像素值，如果小于 0，则 0 为处理后的像素值。Contrast 为对比度系数，其取值范围为 $[0 \sim 2]$ 。

其中 Prgb 表示处理之前的像素值，Crgb 为第一步计算以后的中间结果。

实现图像对比度调整的代码如下：

```

package com.book.chapter.four;

import java.awt.image.BufferedImage;

public class ContrastFilter extends AbstractBufferedImageOp {

    private float contrast;

    public ContrastFilter()
    {

```

```

        this(0.0f);
    }

    // calculate contrast
    public ContrastFilter(float c)
    {
        this.contrast = c;
    }

    public float getContrast() {
        return contrast;
    }

    public void setContrast(float contrast) {
        this.contrast = contrast;
    }

    @Override
    public BufferedImage filter(BufferedImage src,
                               BufferedImage dest) {
        int width = src.getWidth();
        int height = src.getHeight();

        if (dest == null)
            dest = createCompatibleDestImage(src, null);

        int[] inPixels = new int[width * height];
        int[] outPixels = new int[width * height];
        src.getRGB(0, 0, width, height, inPixels, 0, width);

        // handle user input parameter-contrast
        if(this.contrast > 100)
        {
            contrast = 100;
        }
        if(this.contrast < -100)
        {
            contrast = -100;
        }
        contrast = (1 + contrast / 100.0f);

        // adjust image contrast pixel by pixel, here
        int index = 0;
        for (int row = 0; row < height; row++) {
            int ta = 0, tr = 0, tg = 0, tb = 0;
            for (int col = 0; col < width; col++) {
                index = row * width + col;
                ta = (inPixels[index] >> 24) & 0xff;
                tr = (inPixels[index] >> 16) & 0xff;
                tg = (inPixels[index] >> 8) & 0xff;
                tb = inPixels[index] & 0xff;
            }
        }
    }

```

```

// make it more difference
float cr = ((tr / 255.0f) - 0.5f) * contrast;
float cg = ((tg / 255.0f) - 0.5f) * contrast;
float cb = ((tb / 255.0f) - 0.5f) * contrast;

// output RGB value
tr = (int)((cr + 0.5f) * 255.0f);
tg = (int)((cg + 0.5f) * 255.0f);
tb = (int)((cb + 0.5f) * 255.0f);

// write it back
outPixels[index] = (ta << 24) | (clamp(tr) << 16)
                  | (clamp(tg) << 8) | clamp(tb);
}
}
setRGB(dest, 0, 0, width, height, outPixels);
return dest;
}

public int clamp(int value) {
    return value > 255 ? 255 :
           (value < 0 ? 0 : value);
}
}

```

同样，测试该功能只需要调用第3章提到的 ImagePanel 类的 process 方法即可。

4.7 综合应用——调整图像亮度、对比度和饱和度

上面几节着重介绍了如何调整图像亮度、对比度、饱和度，接触了日常图像处理中最重要的几个属性调整方法。从上面的介绍也不难发现，调整图像亮度、对比度的方法有很多，不能简单以好坏评价，调整图像饱和度的方法其实也不少，但是基本都是基于 Hue 色彩空间完成的。本节将上面几节所学知识融合到一起，实现一个可以运行的、调整图像亮度、对比度、饱和度的 Swing 应用程序。首先介绍实现思路，在这里用一个类来实现这三个功能，所以首先要做到的就是对输入参数的支持与处理。其次，在 UI 编程方面，要支持用户数据输入，这里使用了三个 JSlider 滑块组件来对应亮度、对比度、饱和度的调整参数，调整完成后单击【确定】即可生效。

1. UI 编程实现思路

第3章中介绍了一个测试图像处理的 Swing 程序，这里只要对此程序稍加修改，就可以实现亮度、对比度和饱和度这三个参数的用户输入。做法很简单，只要在单击【处理】按钮的响应事件中弹出一个对话框，其中包含三个 JSlider 组件（滑块），分别对应亮度、对比度、饱和度三个属性调节的值范围，拖动滑块调整三个值，然后单击【确定】按钮即可得到效果。

对话框实现的代码如下：

```
package com.book.chapter.four;

import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.FlowLayout;
import java.awt.GridLayout;
import java.awt.Toolkit;
import java.awt.Window;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JDialog;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JSlider;

public class BrightContrastSatUI extends JDialog {

    /**
     *
     */
    private static final long serialVersionUID = 1L;
    private JButton okBtn;
    private JLabel bLabel;
    private JLabel cLabel;
    private JLabel sLabel;
    private JSlider bSlider;
    private JSlider cSlider;
    private JSlider sSlider;

    public BrightContrastSatUI(JFrame parent)
    {
        super(parent, "调整图像亮度、对比度、饱和度");
        initComponents();
    }

    private void initComponents() {
        okBtn = new JButton("确定");
        bLabel = new JLabel("亮度");
        cLabel = new JLabel("对比度");
        sLabel = new JLabel("饱和度");
        bSlider = new JSlider(JSlider.HORIZONTAL, -100, 100, 0);
        bSlider.setMajorTickSpacing(40);
        bSlider.setMinorTickSpacing(10);
        bSlider.setPaintLabels(true);
        bSlider.setPaintTicks(true);
        bSlider.setPaintTrack(true);
```

```

cSlider = new JSlider(JSlider.HORIZONTAL, -100, 100, 0);
cSlider.setMajorTickSpacing(40);
cSlider.setMinorTickSpacing(10);
cSlider.setPaintLabels(true);
cSlider.setPaintTicks(true);
cSlider.setPaintTrack(true);

sSlider = new JSlider(JSlider.HORIZONTAL, -100, 100, 0);
sSlider.setMajorTickSpacing(40);
sSlider.setMinorTickSpacing(10);
sSlider.setPaintLabels(true);
sSlider.setPaintTicks(true);
sSlider.setPaintTrack(true);

this.getContentPane().setLayout(new BorderLayout());
JPanel bPanel = new JPanel();
bPanel.setLayout(new FlowLayout(FlowLayout.CENTER));
bPanel.add(bLabel);
bPanel.add(bSlider);

JPanel cPanel = new JPanel();
cPanel.setLayout(new FlowLayout(FlowLayout.CENTER));
cPanel.add(cLabel);
cPanel.add(cSlider);

JPanel sPanel = new JPanel();
sPanel.setLayout(new FlowLayout(FlowLayout.CENTER));
sPanel.add(sLabel);
sPanel.add(sSlider);

JPanel contentPanel = new JPanel();
contentPanel.setLayout(new GridLayout(3,1));
contentPanel.add(bPanel);
contentPanel.add(cPanel);
contentPanel.add(sPanel);

JPanel btnPanel = new JPanel();
btnPanel.setLayout(new FlowLayout(FlowLayout.RIGHT));
btnPanel.add(okBtn);
this.getContentPane().add(contentPanel, BorderLayout.CENTER);
this.getContentPane().add(btnPanel, BorderLayout.SOUTH);
this.pack();
}

public static void centre(Window w) {
    Dimension us = w.getSize();
    Dimension them = Toolkit.getDefaultToolkit().getScreenSize();
    int newX = (them.width - us.width) / 2;
    int newY = (them.height - us.height) / 2;
    w.setLocation(newX, newY);
}

```

```

public int getBright()
{
    return bSlider.getValue();
}

public int getContrast()
{
    return cSlider.getValue();
}

public int getSaturation()
{
    return sSlider.getValue();
}

public void showUI()
{
    centre(this);
    this.setVisible(true);
}

public void setupActionListener(ActionListener l)
{
    this.okBtn.addActionListener(l);
}
}

```

2. 亮度、对比度、饱和度调整编程思路

调整饱和度与亮度有一定的直接联系，饱和度越高颜色越亮，否则越暗淡，而亮度越高则图像越发白，亮度越低则图像越发黑，对比度越明显则图像细节越突出，反之则细节不是很明显。知道这些知识以后，相信你已清楚不要同时调整图像的亮度与饱和度两个属性了。而图像对比度调整相对比较独立，可以与调整图像的亮度或者饱和度同时进行。在编程实现中，为了代码重用，首先把像素值在 HSL 与 RGB 色彩空间互换，放到抽象类 `AbstractBufferedImageOp` 中，做成两个公共方法，这样让所有继承类都可以使用，其次把检查 RGB 值范围的方法也放到抽象类 `AbstractBufferedImageOp` 中，让所有的子类/继承类都可以使用。完成以后的调整饱和度、亮度、对比度的完整源代码如下：

```

package com.book.chapter.four;

import java.awt.image.BufferedImage;

public class BCSTAdjustFilter extends AbstractBufferedImageOp {
    private double contrast;
    private double brightness;
    private double saturation;
}

```

```

public double getContrast() {
    return contrast;
}

public void setContrast(double contrast) {
    this.contrast = contrast;
}

public double getBrightness() {
    return brightness;
}

public void setBrightness(double brightness) {
    this.brightness = brightness;
}

public double getSaturation() {
    return saturation;
}

public void setSaturation(double saturation) {
    this.saturation = saturation;
}

@Override
public BufferedImage filter(BufferedImage src, BufferedImage dest) {
    handleParameters();
    int width = src.getWidth();
    int height = src.getHeight();
    if (dest == null)
        dest = createCompatibleDestImage(src, null);

    int[] inPixels = new int[width*height];
    int[] outPixels = new int[width*height];
    getRGB(src, 0, 0, width, height, inPixels);
    int index = 0;
    for(int row=0; row<height; row++) {
        int ta = 0, tr = 0, tg = 0, tb = 0;
        for(int col=0; col<width; col++) {
            index = row * width + col;
            ta = (inPixels[index] >> 24) & 0xff;
            tr = (inPixels[index] >> 16) & 0xff;
            tg = (inPixels[index] >> 8) & 0xff;
            tb = inPixels[index] & 0xff;
            double[] hsl = rgb2Hsl(new int[]{tr, tg, tb});

            // adjust saturation.
            hsl[1] = hsl[1] * saturation;
            if (hsl[1] < 0.0) {

```

```

        hsl[1] = 0.0;
    }
    if( hsl[1] > 255.0) {
        hsl[1] = 255.0;
    }

    // adjust brightness
    hsl[2] = hsl[2] * brightness;
    if( hsl[2] < 0.0) {
        hsl[2] = 0.0;
    }
    if( hsl[2] > 255.0) {
        hsl[2] = 255.0;
    }

    // back to RGB space
    int[] rgb = hsl2RGB(hsl);
    tr = clamp(rgb[0]);
    tg = clamp(rgb[1]);
    tb = clamp(rgb[2]);

    // adjust contrast
    double cr = ((tr /255.0d) - 0.5d) * contrast;
    double cg = ((tg /255.0d) - 0.5d) * contrast;
    double cb = ((tb /255.0d) - 0.5d) * contrast;

    // output RGB value
    tr = (int)((cr + 0.5f) * 255.0f);
    tg = (int)((cg + 0.5f) * 255.0f);
    tb = (int)((cb + 0.5f) * 255.0f);

    // write it back
    outPixels[index] = (ta << 24) |
        (clamp(tr) << 16)
        | (clamp(tg) << 8) |
        clamp(tb);
    }
}

setRGB( dest, 0, 0, width, height, outPixels );
return dest;
}

private void handleParameters() {
    contrast = (1.0 + contrast/100.0);
    brightness = (1.0 + brightness/100.0);
    saturation = (1.0 + saturation/100.0);
}
}

```

3. 参数取值范围

在 UI 上看到三个参数的取值范围均为 $[-100, 100]$ ，其中 0 表示不改变原值，小于 0 表示亮度 / 对比度 / 饱和度值相比于调整之前降低，反之则提高。

4. 测试

只需要在原来第 3 章的 `actionPerformed()` 方法中加上如下的代码片段即可完成测试。

```
final BrightContrastSatUI bcsUI = new
    BrightContrastSatUI(this);
bcsUI.setupActionListener(new ActionListener(){

    @Override
    public void actionPerformed(ActionEvent e) {
        bcsUI.setVisible(false);
        bcsUI.dispose();
        double s = bcsUI.getSaturation();
        double b = bcsUI.getBright();
        double c = bcsUI.getContrast();
        imagePanel.process(new double[]{s, b, c});
        imagePanel.repaint();
    }
});
bcsUI.showUI();
```

完整源代码请参考下载的源文件。本节其余的源代码也请参照相关源文件，源代码也是本书的一部分，强烈建议详细阅读。在这里也说明一下，笔者喜欢用英文注释，注释所用的英文都很简单，大家都应该可以看得明白。

4.8 小结

本章详细介绍了图像的一些基本属性及其获取方法、图像常见色彩空间，以及相互间转换方法。此外，还介绍了亮度、对比度、饱和度的概念，以及调整它们的方法，此外还介绍了一些编程技巧。需要特别说明的是调整图像亮度、对比度、饱和度的方法有很多，这里介绍的方法只是其中之一，感兴趣的读者自己可以进一步编程实践，尝试更多属于自己的方式，为程序打上自己的印记。这也是笔者一直推崇的学习方法，只有不断地编程实践才能更好地促进对理论知识的理解与灵活应用。



像素基本操作

第4章介绍了图像的常见属性，特别介绍了其中经常需要调整的三个基本属性，即图像的亮度、对比度、饱和度。通过学习如何调整这三个属性，相信读者对图像处理中的像素操作有了最基本的认知，明白了图像像素是图像组成中最基本也是最重要的数据单元，本章将继续学习关于图像像素的基本操作，并且通过一些非常实用的图像颜色特效来演示像素基本操作在图像处理中的巧妙运用。

首先基于自然色彩滤镜应用实践来剖析像素基本操作——加减乘除对一幅图像的影响，然后通过两幅图像之间的像素操作来介绍透明通道混合（alpha-blending）的常用方法，最后通过实现图像立体水印文字来说明像素基本操作在现实世界的运用。所有这些知识都是立足于实际问题来阐述的，真正做到学以致用，注重编程与实践、科学与趣味性并重。

5.1 大自然的色彩——自然系列滤镜

在很多移动或在线的图像处理 App 中，常会看到将一张照片变换为一系列色彩不同的照片，给这些色彩不同的照片取的名字也是五花八门，什么都有。本节通过一些简单像素操作，得到不同色彩的图像。笔者给这一系列功能也取了名字，叫自然系列滤镜。好吧，言归正传，让我们来看看如何实现这一系列色彩不同滤镜的编程。

本节实现的自然系列滤镜有以下9种，都是对像素做加减乘除简单运算以后获得的。

□ 空气风格

对图像的每个像素值在 R、G、B 三个分量上分别两两相加，然后取平均值作为新像素值，实现代码如下（tr、tg、tb 表示输入 RGB 像素值三个分量，pixel 表示处理后的像素值）：

```

pixel[1] = (tg + tb) / 2;
pixel[2] = (tr + tb) / 2;
pixel[3] = (tg + tr) / 2;

```

□ 燃情风格

实现更为简单，对 RGB 像素值三个分量进行简单的求取平均值后再处理即可，实现代码如下：

```

int gray = (tr + tg + tb) / 3;
pixel[1] = clamp(gray * 3);
pixel[2] = gray;
pixel[3] = gray / 3;

```

□ 雾风格

通过建立颜色查找表实现，目的是减少计算量，同时这也是很多图像色彩调整的技巧之一，在本节的后面将会详细介绍，雾风格的代码如下 (fogLookup 为颜色查找表)：

```

pixel[1] = fogLookup[tr];
pixel[2] = fogLookup[tg];
pixel[3] = fogLookup[tb];

```

□ 冰冻风格

属于冷色调的调色，所以饱和度与亮度应该相对降低，其代码如下：

```

pixel[1] = clamp((int)Math.abs((tr - tg - tb) * 1.5));
pixel[2] = clamp((int)Math.abs((tg - tb - pixel[1]) * 1.5));
pixel[3] = clamp((int)Math.abs((tb - pixel[1] - pixel[2]) * 1.5));

```

□ 熔岩风格

也是先将像素值三个分量相加求取平均值以后再进行简单的运算即可，代码如下：

```

int gray = (tr + tg + tb) / 3;
pixel[1] = gray;
pixel[2] = Math.abs(tb - 128);
pixel[3] = Math.abs(tb - 128);

```

□ 金属风格

金属风格色彩比较单一，但是轮廓分明，有点灰暗的感觉，通过获取 RGB 分量中 R 分量来取值处理实现，当然也可以通过其他分量来实现，处理方法类似，代码如下：

```

float r = Math.abs(tr - 64);
float g = Math.abs(r - 64);
float b = Math.abs(g - 64);
float gray = ((222 * r + 707 * g + 71 * b) / 1000);
r = gray + 70;
r = r + (((r - 128) * 100) / 100f);
g = gray + 65;
g = g + (((g - 128) * 100) / 100f);
b = gray + 75;

```

```

b = b + ((b - 128) * 100) / 100f);
pixel[1] = clamp((int)r);
pixel[2] = clamp((int)g);
pixel[3] = clamp((int)b);

```

□ 海洋风格

大海总是蔚蓝色的，所以海洋风格就是把像素值 RGB 的三个分量相加取得平均数，然后将最大的比重放到蓝色分量中，代码如下：

```

int gray = (tr + tg + tb) / 3;
Vpixel[1] = clamp(gray / 3);
pixel[2] = gray;
pixel[3] = clamp(gray * 3);

```

□ 湖水风格

湖水的颜色总是略微带有神秘色彩的意味，代码实现如下：

```

int gray = (tr + tg + tb) / 3;
pixel[1] = clamp(gray - tg - tb);
pixel[2] = clamp(gray - pixel[1] - tb);
pixel[3] = clamp(gray - pixel[1] - pixel[2]);

```

□ 彩虹风格

通过颜色查找表实现，根据一张彩虹图片初始化生成彩虹颜色查找表。关于如何生成查找表稍后会详细介绍，以下代码主要演示使用查找表来实现彩虹风格。

```

pixel[1] = rainbowLookup[tr];
pixel[2] = rainbowLookup[tg];
pixel[3] = rainbowLookup[tb];

```

通过上述代码详细演绎像素之间加减乘除的巧妙运用，实现了各种不同颜色风格的图像调整，上述实现中还涉及另外一个重要的图像处理技巧——建立图像颜色查找表，在彩虹风格与雾风格中，分别通过两种不同的初始化查找表方法建立了图像查找表，在风格处理中，只要将图像像素作为索引值获取查找表中的值即为新的像素值，大致来说图像查找表建立可以基于以下两种方式。

第一种方式是基于规则或特定算法生成颜色查找表，雾风格的颜色查找表的建立就是基于这种方式的，代码如下：

```

private void buildFogLookupTable() {
    fogLookup = new int[256];
    int fogLimit = 40;
    for(int i=0; i<fogLookup.length; i++)
    {
        if(i > 127)
        {
            fogLookup[i] = i - fogLimit;
            if(fogLookup[i] < 127)

```

```

    {
        fogLookup[i] = 127;
    }
    else
    {
        fogLookup[i] = i + fogLimit;
        if(fogLookup[i] > 127)
        {
            fogLookup[i] = 127;
        }
    }
}
}

```

第二种方式是基于特定图像给出的颜色作为标准来生成颜色查找表，彩虹风格就基于这种方式，代码如下：

```

private void buildRainBowLookupTable()
{
    rainbowLookup = new int[256];
    java.net.URL imageURL = this.getClass().
        getResource("rainbow.png");
    try {
        BufferedImage image = ImageIO.read(imageURL);
        int width = image.getWidth();
        int height = image.getHeight();
        int[] inPixels = new int[width * height];
        getRGB(image, 0, 0, width, height, inPixels);
        for (int col = 0; col < width; col++) {
            rainbowLookup[col] = inPixels[col];
        }
    } catch (IOException e) {
        System.err.println("An error occured " +
            "when loading the image icon...");
    }
}

```

这里只是简单地列举一下图像处理中颜色查找表的建立方法，在细节上还有很多可以改进的地方，这些内容将在后面的章节中不断深入，这里只是先引入概念，达到“授人以鱼不如授人以渔”的效果。本节的全部源代码可以参见源文件，源代码也是本书的一部分，请读者一定仔细阅读，动手实践。

细心的读者可能也注意到了，本节对图像的处理只涉及了一些最基本的数学运算，下一节将详细剖析图像像素加减乘除运算的实际意义。

5.2 图像像素加减乘除

通过上一节的学习，大家对图像像素的基本运算有了一定的了解，这里通过实例更加深

入地学习图像像素加减乘除的实际效果。

1. 像素加操作

对于像素加操作，首先想到的是对每个像素加上特定值，如果用公式表示，则为：

$$P' = P + N$$

其中 P' 表示处理以后的像素值， P 表示原像素值， N 表示噪声，直白一点讲也可以是一个值或一个数学表达式。这里通过一个随机数函数来生成一些随机值加到每个像素值上，从而达到图像加噪效果。像素加操作的代码实现如下：

```
private int addNoise(int p)
{
    boolean valid = false;
    do {
        int ran = (int)Math.round(
            rnd.nextGaussian()*range);
        int v = p + ran; // pixel add noise
        valid = v>=0 && v<=255;
        if (valid) p = v;
    } while (!valid);

    return p;
}
```

2. 像素减操作

从本质上来说，像素减操作也是图像像素加操作，这里以当前像素点像素值与前一个像素点像素值相减得到的值重新赋值作为当前像素点像素值，大致的数学表达可以为：

$$P1' = P1 - P0$$

其中 $P1$ 表示当前像素， $P0$ 表示当前像素在 X 轴方向的左侧相邻像素， $P1'$ 表示新像素值。如图 5-1 所示。

该像素操作的代码实现如下：

```
private int[] minus(int[] rgb, int p) {
    int tr = (p >> 16) & 0xff;
    int tg = (p >> 8) & 0xff;
    int tb = p & 0xff;
    rgb[0] = clamp(rgb[0] - tr);
    rgb[1] = clamp(rgb[1] - tg);
    rgb[2] = clamp(rgb[2] - tb);
    return rgb;
}
```

P0	P1	

$$P1' = P1 - P0$$

图 5-1 像素减操作

注意 两个像素值相减，得到的可能是负数，所以 `clamp` 方法用来处理越界像素值，RGB 像素的各个分量取值均为 0 ~ 255。

3. 像素乘操作

这里也通过一个实际图像处理效果——聚光灯效果来演示像素的乘法操作。顾名思义，聚光灯效果就是将一束光打到一张图像的正中心，从中心开始到边缘，随着距离变化，亮度不断变暗，直至图像边缘。这里有以下两个重要的计算步骤。

1) 计算从中心到边缘变换时每个像素要乘的系数，中心点的系数为1，边缘的系数为0，随着梯度的变化，系数值由像素坐标点 $P(x, y)$ 到中心点 $C(x, y)$ 的距离决定。首先要计算中心点到边缘的最大距离值，即 P 作为 $P(0,0)$ 到中心点 $C(x, y)$ 的距离，其中 x 与 y 分别为图像宽度与高度的一半，根据两点之间欧几里得几何距离公式：

$$Distance = \sqrt{(X2 - X1) \times (X2 - X1) + (Y2 - Y1) \times (Y2 - Y1)}$$

可以首先得到最大距离 D_{\max} ，然后根据任意像素点 P 与中心点的距离 D ，计算出系数 $factor$ ：

$$factor = 1 - (D/D_{\max})$$

2) 根据第一步计算出来的系数求得新的像素值，即为：

$$P' = P \times factor$$

大致的代码实现如下：

```
private int[] multiple(int[] rgb, double maxDistance,
    int cx, int cy, int row, int col) {
    double scale = 1.0 -
        getDistance(cx, cy, col, row)/maxDistance;
    scale = scale * scale;
    rgb[0] = (int)(scale * rgb[0]);
    rgb[1] = (int)(scale * rgb[1]);
    rgb[2] = (int)(scale * rgb[2]);
    return rgb;
}
```

从运行效果可以看出，图像从中心开始到边缘随距离变化，图像慢慢变暗。

4. 像素除操作

像素除操作本质上就是像素乘操作，这里就不再赘述举例。

本节提到内容的完整源代码如下：

```
package com.book.chapter.five;

import java.awt.image.BufferedImage;
import java.util.Random;

import com.book.chapter.four.AbstractBufferedImageOp;
/**
 * plus/minus/multiplication/division
 * @author fish
 *
 */
public class PMMDFilter extends AbstractBufferedImageOp {
```



```

public final static int PLUS = 1;
public final static int MINUS = 2;
public final static int MULTIPLE = 4;

private Random rnd;
private double range;
private int type;

public PMMDFilter()
{
    type = MULTIPLE;
    rnd = new Random();
    range = 25.0;
}

@Override
public BufferedImage filter(BufferedImage src,
    BufferedImage dest) {
    int width = src.getWidth();
    int height = src.getHeight();

    if (dest == null)
        dest = createCompatibleDestImage(src, null);

    int[] inPixels = new int[width * height];
    int[] outPixels = new int[width * height];
    getRGB(src, 0, 0, width, height, inPixels);
    int index = 0;
    int centerX = width/2;
    int centerY = height/2;
    double maxDistance = Math.sqrt(centerX * centerX +
        centerY * centerY);
    for (int row = 0; row < height; row++) {
        int ta = 0, tr = 0, tg = 0, tb = 0;
        for (int col = 0; col < width; col++) {
            index = row * width + col;
            ta = (inPixels[index] >> 24) & 0xff;
            tr = (inPixels[index] >> 16) & 0xff;
            tg = (inPixels[index] >> 8) & 0xff;
            tb = inPixels[index] & 0xff;
            int[] rgb = new int[] {tr, tg, tb};
            // plus
            if (type == PLUS)
            {
                rgb = plus(rgb);
            }
            // minus
            if (type == MINUS)
            {
                int pcol = col - 1;
                if (pcol < 0 || pcol >= width)

```

```

    {
        pcol = 0;
    }
    int index2 = row * width + pcol;
    rgb = minus(rgb, inPixels[index2]);
}
if(type == MULTIPLE)
{
    rgb = multiple(rgb, maxDistance,
        centerX, centerY, row, col);
}
tr = rgb[0];
tg = rgb[1];
tb = rgb[2];
outPixels[index] = (ta << 24) | (tr << 16) | (tg << 8) | tb;
}
}
setRGB(dest, 0, 0, width, height, outPixels);
return dest;
}

```

```

private int[] multiple(int[] rgb, double maxDistance,
    int cx, int cy, int row, int col) {

```

```

    double scale = 1.0 -
        getDistance(cx, cy, col, row)/maxDistance;

```

```

    scale = scale * scale;

```

```

    rgb[0] = (int)(scale * rgb[0]);

```

```

    rgb[1] = (int)(scale * rgb[1]);

```

```

    rgb[2] = (int)(scale * rgb[2]);

```

```

    return rgb;
}

```

```

private int[] minus(int[] rgb, int p) {

```

```

    int tr = (p >> 16) & 0xff;

```

```

    int tg = (p >> 8) & 0xff;

```

```

    int tb = p & 0xff;

```

```

    rgb[0] = clamp(rgb[0] - tr);

```

```

    rgb[1] = clamp(rgb[1] - tg);

```

```

    rgb[2] = clamp(rgb[2] - tb);

```

```

    return rgb;
}

```

```

private int[] plus(int[] rgb)

```

```

{
    rgb[0] = addNoise(rgb[0]);

```

```

    rgb[1] = addNoise(rgb[1]);

```

```

    rgb[2] = addNoise(rgb[2]);

```

```

    return rgb;
}

```

```
private int addNoise(int p)
{
    boolean valid = false;
    do {
        int ran = (int)Math.round(
            rnd.nextGaussian()*range);
        int v = p + ran;// pixel add noise
        valid = v>=0 && v<=255;
        if (valid) p = v;
    } while (!valid);

    return p;
}

private double getDistance(int cx, int cy,
    int px, int py) {
    double xx = (cx - px)*(cx - px);
    double yy = (cy - py)*(cy - py);
    return (int)Math.sqrt(xx + yy);
}
}
```

测试该程序，同样只要在第 3 章的 ImagePanel 类的 process 方法中完成如下代码：

```
public void process()
{
    PMMDFilter filter = new PMMDFilter();
    destImage = filter.filter(sourceImage, null);
}
```

然后运行 MainUI.java 方法，选择图片，单击【处理】按钮即可。

5.3 两幅图像的融合与叠加

本节通过讲述两幅图像的透明通道融合与叠加实践来演示两幅图像像素之间的加减乘除，然后就可以得到一幅融合了两幅图像内容的新图片。下面会仔细剖析几种典型图像叠加方法，继续演示图像像素的基本操作。在开始之前，为了简便计算，首先假设两幅图像的大小完全一致，对应的像素数组分别为 A 与 B ，对应的任意单个像素值分别为 a 与 b ，混合以后的像素值为 c 。

□ 乘法叠加

乘法叠加的公式可以表示为 $c = (a \times b) / 255$ ，对于 RGB 像素值，代码实现如下：

```
private int modeOne(int v1, int v2) {
    return (v1 * v2) / 255;
}
```

□ 加法叠加

加法叠加的公式可以表示为 $c = (a + b)/2$, 对于 RGB 像素值, 代码实现如下:

```
private int modeTwo(int v1, int v2) {
    return (v1 + v2) / 2;
}
```

□ 减法叠加

减法叠加的公式可以表示为 $c = |a - b|$, 就是取它们的差值, 从公式看出基于该方法叠加混合两种图像, 更加突出与强调两张图像中对应像素的不同。代码实现如下:

```
private int modeThree(int v1, int v2) {
    return Math.abs(v1 - v2);
}
```

□ 取反叠加

取反叠加的公式可以表示为 $c = 255 - ((255 - a) \times (255 - b) / 255)$, 首先对各自的像素值取反, 然后使用乘法叠加之后对得到的结果再次取反, 这里说的像素值取反是指对任意原像素值 p 来说, 它的取反结果为 $255 - p$ 。取反叠加的代码实现如下:

```
private int modeFour(double v1, double v2) {
    double p = (int)((255 - v1) * (255 - v2));
    return (int)(255 - (p / 255));
}
```

□ 加法取反叠加

加法取反叠加的公式可以表示为 $c = 255 - (a + b) \{ (a + b) < 255 \} | c = 0 \{ (a + b) \geq 255 \}$, 该表达式表示如果 $a + b$ 的和小于 255, 则取前面部分作为计算结果, 否则 $c = 0$ 。代码实现如下:

```
private int modeFive(double v1, double v2) {
    int p = (int)(v1 + v2);
    if(p > 255)
        return 0;
    else
        return 255 - p;
}
```

□ 除法取反叠加

除法取反叠加的公式表示为 $c = (a / (255 - b)) \times 255$, 其中 b 不能等于 255 (读者可以自己想一下其原因), 在 $b = 255$ 时, 令 $c = 255$ 。代码实现如下:

```
private int modeSix(double v1, double v2) {
    if(v2 == 255)
        return 0;
    double p = (v1 / (255 - v2)) * 255;
    return clamp((int)p);
}
```

上面 6 种图像混合方式的完整代码如下：

```
package com.book.chapter.five;

import java.awt.image.BufferedImage;

import com.book.chapter.four.AbstractBufferedImageOp;

public class BlendFilter extends AbstractBufferedImageOp {

    public final static int MULTIPLY_PIXEL = 1;
    public final static int PLUS_PIXEL = 2;
    public final static int MINUS_PIXEL = 3;
    public final static int INVERSE_PIXEL = 4;
    public final static int INVERSE_PLUS_PIXEL = 5;
    public final static int DIVISION_PIXEL = 6;

    private int MODE;
    private BufferedImage secondImage;

    public BlendFilter() {
        MODE = MULTIPLY_PIXEL;
    }

    public void setBlendMode(int mode) {
        this.MODE = mode;
    }

    public void setSecondImage(BufferedImage image) {
        this.secondImage = image;
    }

    @Override
    public BufferedImage filter(BufferedImage src, BufferedImage dest) {
        checkImages(src);
        int width = src.getWidth();
        int height = src.getHeight();

        if (dest == null)
            dest = createCompatibleDestImage(src, null);

        int[] input1 = new int[width * height];
        int[] input2 = new int[secondImage.getWidth() * secondImage.getHeight()];
        int[] outPixels = new int[width * height];
        getRGB(src, 0, 0, width, height, input1);
        getRGB(secondImage, 0, 0, secondImage.getWidth(),
            secondImage.getHeight(), input2);
        int index = 0;
        int tal = 0, trl = 0, tgl = 0, tbr = 0;
        for (int row = 0; row < height; row++) {
            for (int col = 0; col < width; col++) {
```

```

index = row * width + col;
tal = (input1[index] >> 24) & 0xff;
trl = (input1[index] >> 16) & 0xff;
tgl = (input1[index] >> 8) & 0xff;
tbl = input1[index] & 0xff;
int[] rgb = getBlendData(trl, tgl, tbl, input2, row, col);
outPixels[index] = (tal << 24) | (rgb[0] << 16) | (rgb[1] << 8)
| rgb[2];
}
}

```

5.4 一个更加深入的案例——图像上覆盖文字效果

```

setRGB(dest, 0, 0, width, height, outPixels);
return dest;
}

private int[] getBlendData(int trl, int tgl, int tbl, int[] input, int row,
int col) {
int width = secondImage.getWidth();
int height = secondImage.getHeight();
if (col >= width || row >= height) {
return new int[] { trl, tgl, tbl };
}

int index = row * width + col;
int tr = (input[index] >> 16) & 0xff;
int tg = (input[index] >> 8) & 0xff;
int tb = input[index] & 0xff;
int[] rgb = new int[3];
if (MODE == MULTIPLY_PIXEL) {
rgb[0] = modeOne(trl, tr);
rgb[1] = modeOne(tgl, tg);
rgb[2] = modeOne(tbl, tb);
} else if (MODE == PLUS_PIXEL) {
rgb[0] = modeTwo(trl, tr);
rgb[1] = modeTwo(tgl, tg);
rgb[2] = modeTwo(tbl, tb);
} else if (MODE == MINUS_PIXEL) {
rgb[0] = modeThree(trl, tr);
rgb[1] = modeThree(tgl, tg);
rgb[2] = modeThree(tbl, tb);
} else if (MODE == INVERSE_PIXEL) {
rgb[0] = modeFour(trl, tr);
rgb[1] = modeFour(tgl, tg);
rgb[2] = modeFour(tbl, tb);
} else if (MODE == INVERSE_PLUS_PIXEL) {
rgb[0] = modeFive(trl, tr);
rgb[1] = modeFive(tgl, tg);
rgb[2] = modeFive(tbl, tb);
} else if (MODE == DIVISION_PIXEL) {
rgb[0] = modeSix(trl, tr);
rgb[1] = modeSix(tgl, tg);
rgb[2] = modeSix(tbl, tb);
}

return rgb;
}

```


上面 3 种图像混合方式的完整代码如下：

```
private int modeOne(int v1, int v2) {
    return (v1 * v2) / 255;
}

private int modeTwo(int v1, int v2) {
    return (v1 + v2) / 2;
}

private int modeThree(int v1, int v2) {
    return Math.abs(v1 - v2);
}

private int modeFour(double v1, double v2) {
    double p = (int) ((255 - v1) * (255 - v2));
    return (int) (255 - (p / 255));
}

private int modeFive(double v1, double v2) {
    int p = (int) (v1 + v2);
    if (p > 255)
        return 0;
    else
        return 255 - p;
}

private int modeSix(double v1, double v2) {
    if (v2 == 255)
        return 0;
    double p = (v1 / (255 - v2)) * 255;
    return clamp((int)p);
}

private void checkImages(BufferedImage src) {
    int width = src.getWidth();
    int height = src.getHeight();
    if (secondImage == null
        || secondImage.getWidth() > width
        || secondImage.getHeight() > height) {
        throw new IllegalArgumentException(
            "the width, height of the input image must be " +
            "great than blend image");
    }
}
```

运行测试该实例程序，同样只使用第 3 章中提到的 `ImagePanel` 类 `process` 方法，调用该实例类即可实现。

基于像素操作的图像混合与叠加算法有很多，几乎主流的图像处理软件中都提供两幅图

像叠加与混合的功能,混合叠加方法也非常多,感兴趣的读者可以自己做更加深入的研究。上面演示的6种方法主要是从实例角度说明一些常见算术运算在图像处理过程的运用,帮助读者慢慢建立简单的数学概念。从本章开始将介绍一些基本的数学知识,从本质上来说,图像处理是建立在数学模型的基础上的,实际编程应用都可以从数学模型中找到痕迹。希望读者在编程实践的同时,也适当学习一些基本数学知识的运用技巧。

5.4 一个更加深入的应用实践——图像上轧花文字效果

本节通过一个具有实际意义的文字与图像融合实例来更加深入与系统地学习图像处理中对像素操作的高级技巧。对于任意一张彩色图像,准备好一张黑白单色文字图像,通过一系列的像素操作最终形成彩色图像上的轧花文字,从而实现特殊的文字水印特效图像。这在实际项目中非常有意义,很多大型的互联网站点都是通过显示文字水印来防止图像资源被盗用的。本节的学习可帮助读者打开这方面的思路,提高在工作中解决实际图像处理问题的能力。

1. 基本思路

主要利用文字图像像素在X和Y方向上同时移位一个像素,完成对二值图像提取文字骨架的操作,宽度为一个像素,然后将提取的骨架按照一定的像素值与目标图像的像素值融合叠加即可。在这里为了处理上的简洁,假设文字图像为黑白二值图像,并设定取到的骨架像素为黑色,然后检查文字图像中的黑色像素,根据像素位置得到相对目标图像的像素,两个像素进行直接设置,最后在目标图像上得到一个富有立体感的文字水印。

2. 程序实现步骤

1) 读入准备好的黑白文字图片,创建两张大小一致的单色白板图像(BufferedImage)。

Top-left 位移一个像素,将每个对应的像素值 copy 到单色白板图像中,如图 5-2 所示,其中,虚线内的像素将向上向左移动一个像素。

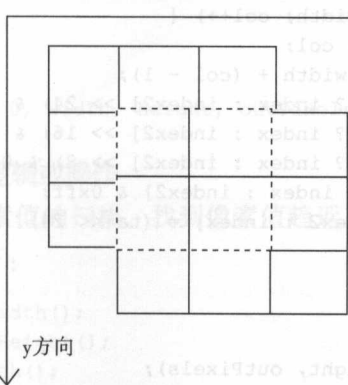


图 5-2 虚线待移动像素

2) 与第 1 步类似, 只是向下向右移动一个像素, 将每个对应的像素值 copy 到另外单色白板图像中。

3) 分别将第 2、3 两步中得到的图片与原来的图像进行逻辑“或”操作, 得到左上与右下的文字骨架。

4) 将两个文字骨架像素填充到目标彩色图像中, 即可得到立体轧花效果水印文字。

3. 编程关键点：像素位移处理

主要是利用像素移位操作实现一个像素宽的整体图像位置移动, 向哪个方向移动通过 boolean 变量 isTop 来实现控制, 这是实现像素移位的关键之处, 相关代码如下:

```
int width = src.getWidth();
int height = src.getHeight();

if (dest == null)
    dest = createCompatibleDestImage(src, null);

int[] inPixels = new int[width * height];
int[] outPixels = new int[width * height];
getRGB(src, 0, 0, width, height, inPixels);
int index = 0;
int index2 = 0;
// initialization outPixels
for (int row = 0; row < height; row++) {
    for (int col = 0; col < width; col++) {
        index = row * width + col;
        outPixels[index] = (255 << 24) | (255 << 16)
            | (255 << 8) | 255;
    }
}

// one pixel transfer
for (int row = 1; row < height; row++) {
    int ta = 0, tr = 0, tg = 0, tb = 0;
    for (int col = 1; col < width; col++) {
        index = row * width + col;
        index2 = (row - 1) * width + (col - 1);
        ta = (inPixels[isTop ? index : index2] >> 24) & 0xff;
        tr = (inPixels[isTop ? index : index2] >> 16) & 0xff;
        tg = (inPixels[isTop ? index : index2] >> 8) & 0xff;
        tb = inPixels[isTop ? index : index2] & 0xff;
        outPixels[isTop ? index2 : index] = (ta << 24)
            | (tr << 16)
            | (tg << 8) | tb;
    }
}

setRGB(dest, 0, 0, width, height, outPixels);
return dest;
```

4. 编程关键点：文字骨架获取

因为这里的文字图片是白色背景和黑色文字，所以骨架提取时，像素值接近 0，这就是我们感兴趣的文字内容，而对于白色背景直接填充即可。相关代码如下：

```
// now get one pixel data
int index = 0;
for (int row = 0; row < height; row++) {
    int ta = 0, tr = 0, tg = 0, tb = 0;
    int ta2 = 0, tr2 = 0, tg2 = 0, tb2 = 0;
    for (int col = 0; col < width; col++) {
        index = row * width + col;
        ta = (inPixels[index] >> 24) & 0xff;
        tr = (inPixels[index] >> 16) & 0xff;
        tg = (inPixels[index] >> 8) & 0xff;
        tb = inPixels[index] & 0xff;

        ta2 = (outPixels[index] >> 24) & 0xff;
        tr2 = (outPixels[index] >> 16) & 0xff;
        tg2 = (outPixels[index] >> 8) & 0xff;
        tb2 = outPixels[index] & 0xff;

        if (tr2 == tr && tg == tg2 && tb == tb2) {
            outPixels[index] = (255 << 24) | (255 << 16)
                | (255 << 8)
                | 255;
        } else {
            if (tr2 < 5 && tg2 < 5 && tb2 < 5) {
                outPixels[index] = (ta2 << 24)
                    | (tr2 << 16)
                    | (tg2 << 8) | tb2;
            } else {
                outPixels[index] = (255 << 24)
                    | (255 << 16)
                    | (255 << 8) | 255;
            }
        }
    }
}
setRGB(destImage, 0, 0, width, height, outPixels);
```

5. 编程关键点：像素逻辑或操作

通过实现对像素点像素值的扫描，找到像素值趋近 0 的文字内容像素点，即可得到想要的轧花结果。相关代码如下：

```
int width = src.getWidth();
int height = src.getHeight();
int dw = dest.getWidth();
int dh = dest.getHeight();
```

```

int[] sinPixels = new int[width * height];
int[] dinPixels = new int[dw * dh];
src.getRGB(0, 0, width, height, sinPixels, 0, width);
dest.getRGB(0, 0, dw, dh, dinPixels, 0, dw);
int index = 0;
int index2 = 0;
for (int y = 0; y < height; y++) {
    for (int x = 0; x < width; x++) {
        index = y * width + x;
        int srgb = sinPixels[index];
        int r1 = (srgb >> 16) & 0xff;
        int g1 = (srgb >> 8) & 0xff;
        int b1 = srgb & 0xff;
        if (r1 > 200 || g1 >= 200 || b1 >= 200) {
            continue;
        }
        index2 = y * dw + x;
        if (colorInverse) {
            r1 = 255 - r1;
            g1 = 255 - g1;
            b1 = 255 - b1;
        }
        dinPixels[index2] = (255 << 24)
            | (r1 << 16) | (g1 << 8) | b1;
    }
}
dest.setRGB(0, 0, dw, dh, dinPixels, 0, dw);

```



注意 关于文字反锯齿：当对图像进行像素处理时，由于很多图像软件在生产黑白文字图片时会自动实现反锯齿功能，所以边缘像素不会是 0 或 255 这样的值，而有一些边缘模糊，且其像素值在 0 ~ 255 之间，所以上面的代码中，当大于 200 时就可以认为是白色背景像素，小于 5 就可以认为是黑色像素，这样就避免了对文字图像进行二值化处理，减少了一些处理步骤。

完整的水印实现程序如下：

```

package com.book.chapter.five;

import java.awt.image.BufferedImage;

import com.book.chapter.four.AbstractBufferedImageOp;

public class BitBlitFilter extends AbstractBufferedImageOp {
    // raster operation - bit block transfer.
    // 1975 for the Smalltalk-72 system, For the Smalltalk-74 system
    private boolean isTop = true;

    /**
     * left - top skeleton or right - bottom.
     */
}

```

```

*
* @param isTop
*/
public void setTop(boolean isTop) {
    this.isTop = isTop;
}

/**
* blend the pixels and get the final output image
*
* @param textImage
* @param targetImage
*/
public void emboss(BufferedImage textImage, BufferedImage targetImage) {
    // BitBlitFilter filter = new BitBlitFilter();
    BufferedImage topImage = filter(textImage, null);
    setTop(false);
    BufferedImage bottomImage = filter(textImage, null);

    int width = textImage.getWidth();
    int height = textImage.getHeight();

    int[] inPixels = new int[width * height];
    int[] outPixels = new int[width * height];
    getRGB(textImage, 0, 0, width, height, inPixels);
    getRGB(topImage, 0, 0, width, height, outPixels);
    processOnePixelWidth(width, height, inPixels, outPixels, topImage);
    getRGB(bottomImage, 0, 0, width, height, outPixels);
    processOnePixelWidth(width, height, inPixels, outPixels, bottomImage);

    // emboss now
    embossImage(topImage, targetImage, true);
    embossImage(bottomImage, targetImage, false);
}

@Override
public BufferedImage filter(BufferedImage src, BufferedImage dest) {
    int width = src.getWidth();
    int height = src.getHeight();

    if (dest == null)
        dest = createCompatibleDestImage(src, null);

    int[] inPixels = new int[width * height];
    int[] outPixels = new int[width * height];
    getRGB(src, 0, 0, width, height, inPixels);
    int index = 0;
    int index2 = 0;
    // initialization outPixels
    for (int row = 0; row < height; row++) {
        for (int col = 0; col < width; col++) {

```



```

        index = row * width + col;
        outPixels[index] = (255 << 24) | (255 << 16) | (255 << 8) | 255;
    }

    // one pixel transfer
    for (int row = 1; row < height; row++) {
        int ta = 0, tr = 0, tg = 0, tb = 0;
        for (int col = 1; col < width; col++) {
            index = row * width + col;
            index2 = (row - 1) * width + (col - 1);
            ta = (inPixels[isTop ? index : index2] >> 24) & 0xff;
            tr = (inPixels[isTop ? index : index2] >> 16) & 0xff;
            tg = (inPixels[isTop ? index : index2] >> 8) & 0xff;
            tb = inPixels[isTop ? index : index2] & 0xff;
            outPixels[isTop ? index2 : index] = (ta << 24) | (tr << 16)
                | (tg << 8) | tb;
        }
        setRGB(dest, 0, 0, width, height, outPixels);
        return dest;
    }

    /**
     * @param width
     * @param height
     * @param inPixels
     * @param outPixels
     * @param destImage
     */
    private void processOnePixelWidth(int width, int height, int[] inPixels,
        int[] outPixels, BufferedImage destImage) {
        // now get one pixel data
        int index = 0;
        for (int row = 0; row < height; row++) {
            int ta = 0, tr = 0, tg = 0, tb = 0;
            int ta2 = 0, tr2 = 0, tg2 = 0, tb2 = 0;
            for (int col = 0; col < width; col++) {
                index = row * width + col;
                ta = (inPixels[index] >> 24) & 0xff;
                tr = (inPixels[index] >> 16) & 0xff;
                tg = (inPixels[index] >> 8) & 0xff;
                tb = inPixels[index] & 0xff;

                ta2 = (outPixels[index] >> 24) & 0xff;
                tr2 = (outPixels[index] >> 16) & 0xff;
                tg2 = (outPixels[index] >> 8) & 0xff;
                tb2 = outPixels[index] & 0xff;

                if (tr2 == tr && tg2 == tg && tb2 == tb) {

```

```

        outPixels[index] = (255 << 24) | (255 << 16) | (255 << 8)
        | 255;
    } else {
        if (tr2 < 5 && tg2 < 5 && tb2 < 5) {
            outPixels[index] = (ta2 << 24) | (tr2 << 16)
            | (tg2 << 8) | tb2;
        } else {
            outPixels[index] = (255 << 24) | (255 << 16)
            | (255 << 8) | 255;
        }
    }
}

setRGB(destImage, 0, 0, width, height, outPixels);
}

/**
 *
 * @param src
 * @param dest
 * @param colorInverse
 * - must be setted here!!!
 */
private void embossImage(BufferedImage src, BufferedImage dest,
    boolean colorInverse) {
    int width = src.getWidth();
    int height = src.getHeight();
    int dw = dest.getWidth();
    int dh = dest.getHeight();

    int[] sinPixels = new int[width * height];
    int[] dinPixels = new int[dw * dh];
    src.getRGB(0, 0, width, height, sinPixels, 0, width);
    dest.getRGB(0, 0, dw, dh, dinPixels, 0, dw);
    int index = 0;
    int index2 = 0;
    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            index = y * width + x;
            int srgb = sinPixels[index];
            int r1 = (srgb >> 16) & 0xff;
            int g1 = (srgb >> 8) & 0xff;
            int b1 = srgb & 0xff;
            if (r1 > 200 || g1 >= 200 || b1 >= 200) {
                continue;
            }
            index2 = y * dw + x;
            if (colorInverse) {
                r1 = 255 - r1;
                g1 = 255 - g1;
                b1 = 255 - b1;
            }

```

```

        outPixels[index] = (255 << 24) | (r1 << 16) | (g1 << 8) | b1;
    } else {
    }
    }
    dest.setRGB(0, 0, dw, dh, dinPixels, 0, dw);
}
}
}

```

运行与测试该程序：

调用此文字水印类，传入文字图片与目标图像以后，执行如下两行代码即可得到最终的效果图像。

```

BitBltFilter filter = new BitBltFilter();
filter.emboss(textImg, targetImg);

```

关于测试代码完整的源程序清单，参见源文件中的索引 BitBltFilterTest.java。

5.5 小结

本章由浅入深地介绍了图像处理中像素算术运算方法与应用技巧，以及其使用实例，对于图像像素的逻辑操作（与、或、非），并没有详细介绍，感兴趣的读者可以自己进一步学习。

像素统计与应用

第5章学习了像素算术运算的各种方法与技巧，一张图像最重要的特征就是各个像素值，对它进行数学建模与变换可以得到各种不同的特征，如图像像素平均值、最大值与最小值、像素个数、像素值的分布等统计信息可以很直观地反映出一张图像的基本特征，本章将会一一介绍如何获取这些统计值，以及如何将这些统计值应用到实际项目中。为了更好地突出统计这些值的方法，在计算统计值时使用的图像都是灰度图像，彩色图像处理只要在RGB三个通道上分别做与灰度一样的计算即可。本章将通过统计像素值出现频率得到图像直方图，以几个实例说明图像直方图在图像二值化、图像均衡化调整、图像匹配等方面的实际应用，为读者进一步打开思路，掌握图像直方图这一重要的统计特征。

同样，在本章中也会介绍一些相关数学知识，这些数学知识的学习与理解对掌握本章内容至关重要，请读者认真对待。再次强调，源代码也是本书的一部分，请务必理解，积极实践。

6.1 统计图像的均值、最大值与最小值

在实际图像处理中图像像素的均值、最大值与最小值是第一步要计算处理的属性数据，本节通过介绍如何计算图像像素的均值、最大值、最小值、方差及其使用技巧，帮助读者学习掌握这一基本知识与应用。

首先来看一下计算图像均值、方差的数学表达：

$$mean = \text{Sum}(P_{x,y}) / (X \times Y)$$

$$stdev = \sqrt{\text{Sum}(P_{x,y} \times P_{x,y}) / (X \times Y) - (mean \times mean)}$$

$$var = stdev \times stdev$$

其中 mean 表示图像像素的算术平均值, stdev 表示标准方差, 标准方差越小说明像素之间的差异越小, 图像像素值与它的算术平均值越接近。上面公式中 $\text{Sum}(P_{x,y})$ 表示图像所有像素之和, X 与 Y 分别表示图像像素宽度与高度。

1. 计算图像像素的均值、最大值、最小值

首先通过下面的程序学习如何计算图像像素的算术平均值、最大值与最小值。代码如下:

```
// calculate mean, MAX, MIN
int max = 0;
int min = 255;
double sum = 0.0;
for (int row = 0; row < height; row++) {
    int tr = 0;
    for (int col = 0; col < width; col++) {
        index = row * width + col;
        tr = (inPixels[index] >> 16) & 0xff;
        min = Math.min(min, tr);
        max = Math.max(max, tr);
        sum += tr;
    }
}
double mean = sum / (width * height);
```

2. 计算图像像素的标准方差

对于计算图像像素的标准方差, 这里使用的方法与常见的方差公式稍有不同, 但是计算更加简洁、方便且实用, 代码如下:

```
// calculate standard deviation
double stdev = 0.0;
double total = width * height;
sum = 0.0;
for (int row = 0; row < height; row++) {
    int tr = 0;
    for (int col = 0; col < width; col++) {
        index = row * width + col;
        tr = (inPixels[index] >> 16) & 0xff;
        sum += tr * tr;
        outPixels[index] = (255 << 24)
            | (tr << 16) | (tr << 8) | tr;
    }
}
stdev = (sum / total) - Math.pow(mean, 2);
```

3. 使用标准方差实现空白图像过滤

在很多图像处理的应用场景中, 由于各种原因, 照相机拍摄的图像很多是没有信息与内容的空白图片, 不一定是黑色或白色, 这些空白图片可能更接近灰度 $[0 \sim 255]$ 之间某个值, 根据这些特点, 我们可以在预处理时先计算图像的方差, 然后将其与给定的某个阈值进行比

较, 如果小于该阈值, 则可以认为是空白图像。这样做正是利用图像方差越小, 则像素之间差异越小的数学原理, 实现了对空白图像的过滤与查找。完整的代码实现如下:

```
package com.book.chapter.six;

import java.awt.image.BufferedImage;
import com.book.chapter.four.AbstractBufferedImageOp;

public class PixelStatisticFilter extends AbstractBufferedImageOp {
    private double threshold;
    private boolean blankImage;

    public PixelStatisticFilter() {
        blankImage = false;
        threshold = 1.0;
    }

    public boolean isBlankImage() {
        return blankImage;
    }

    public double getThreshold() {
        return threshold;
    }

    public void setThreshold(double threshold) {
        this.threshold = threshold;
    }

    @Override
    public BufferedImage filter(BufferedImage src,
        BufferedImage dest) {
        int width = src.getWidth();
        int height = src.getHeight();
        blankImage = false;
        if (dest == null)
            dest = createCompatibleDestImage(src, null);

        int[] inPixels = new int[width * height];
        int[] outPixels = new int[width * height];
        getRGB(src, 0, 0, width, height, inPixels);
        int index = 0;

        // calculate mean, MAX, MIN
        int max = 0;
        int min = 255;
        double sum = 0.0;
        for (int row = 0; row < height; row++) {
            int tr = 0;
            for (int col = 0; col < width; col++) {
```



```

        index = row * width + col;
        tr = (inPixels[index] >> 16) & 0xff;
        min = Math.min(min, tr);
        max = Math.max(max, tr);
        sum += tr;
    }

    double mean = sum / (width * height);

    // calculate standard deviation
    double stdev = 0.0;
    double total = width * height;
    sum = 0.0;
    for (int row = 0; row < height; row++) {
        int tr = 0;
        for (int col = 0; col < width; col++) {
            index = row * width + col;
            tr = (inPixels[index] >> 16) & 0xff;
            sum += tr * tr;
            outPixels[index] = (255 << 24)
                | (tr << 16)
                | (tr << 8) | tr;
        }
    }

    stdev = (sum / total) - Math.pow(mean, 2);
    if (stdev <= threshold)
    {
        blankImage = true;
    }

    System.out.println("均值 = " + mean
        + " 标准方差 = " + stdev);
    setRGB(dest, 0, 0, width, height, outPixels);
    return dest;
}
}

```

测试该程序同样只要在第 3 章的 ImagePanel 的 process 方法内添加如下代码即可：

```

PixelStatisticFilter filter = new PixelStatisticFilter();
destImage = filter.filter(sourceImage, null);
System.out.println("Image Blank is " + filter.isBlankImage());

```

然后在 Eclipse 中运行 MainUI.java，选择一张灰度图像进行测试，即可在控制台看到输出信息与结果。

6.2 灰度图像二值化

本节将在上一节的基础上重点介绍如何使用图像像素的算术平均值实现灰度图像转换为

二值图像,同时还重点介绍一种类似一维 MeanShift 的算法来实现灰度图像转换为二值图像,希望读者通过这两种方法的学习,能够掌握在从灰度到二值的过程中如何正确找到阈值这一关键值,在实现图像二值化的同时保留图像原有信息不丢失。

笔者在 2004 年第一次接触图像编程时,做的第一个图像程序就是把自己的照片变成一张黑白二值图像,用的方法是只要像素值大于 127 就赋值为白色像素 (255),反之则为黑色 (0)。但是当笔者的指导老师问为什么选择 127 作为阈值、依据是什么时,笔者一下子懵了,因为笔者从来没有想过这个问题。显然它不是一种很好的二值化方法,但是直到今天,如果你问一个程序员如何二值化一张图像,他们几乎都会毫不犹豫地告诉你使用该方法。本节学习以后,希望读者能够更专业地回答这个问题。

1. 像素均值作为阈值实现二值化

利用图像算术平均值来实现图像的二值化时,处理步骤大致可以分为如下两步:

- 1) 计算输入图像像素的算术平均值——mean。
- 2) 利用该平均值作为阈值,如果像素值 $P(x, y) > mean$, 令 $P(x, y) = 255$, 否则 $P(x, y) = 0$ 。

实现代码如下:

```
// calculate mean, MAX, MIN
int max = 0;
int min = 255;
double sum = 0.0;
for (int row = 0; row < height; row++) {
    int tr = 0;
    for (int col = 0; col < width; col++) {
        index = row * width + col;
        tr = (pixels[index] >> 16) & 0xff;
        min = Math.min(min, tr);
        max = Math.max(max, tr);
        sum += tr;
    }
}
mean = sum / (width * height);
```

2. 基于一维 MeanShift 方法计算阈值实现二值化

该方法的大致步骤如下:

- 1) 给定一个初始化阈值 T (可以随机生成,或者直接为 127)。
- 2) 根据像素值 $P(x, y)$ 与阈值 T 的比较结果,将其分为对象像素集合 $G1$ 与背景像素集合 $G2$ 。
- 3) 计算 $G1$ 与 $G2$ 的像素平均值 $M1$ 、 $M2$ 。
- 4) 得到新的阈值 $T' = (M1 + M2)/2$ 。
- 5) 比较 T' 与 T 是否相等,如果不等,则令 $T = T'$, 重复第 2 ~ 5 步。
- 6) 最终得到的 T 值作为阈值,像素值大于 T 即为白色,反之为黑色。

使用该方法时，执行到第 5 步还可以通过获取差值来进行计算，如果差值小于一定范围，则停止循环计算新的 T 值。用该方法来寻找阈值相对更加精准，但是计算量较大。

代码实现如下：

```
int inithreshold = 127;
int finalthreshold = 0;
int temp[] = new int[inPixels.length];
for(int index=0; index<inPixels.length; index++) {
    temp[index] = (inPixels[index] >> 16) & 0xff;
}
List<Integer> sub1 = new ArrayList<Integer>();
List<Integer> sub2 = new ArrayList<Integer>();
int means1 = 0, means2 = 0;
while(finalthreshold != inithreshold) {
    finalthreshold = inithreshold;
    for(int i=0; i<temp.length; i++) {
        if(temp[i] <= inithreshold) {
            sub1.add(temp[i]);
        } else {
            sub2.add(temp[i]);
        }
    }
    means1 = getMeans(sub1);
    means2 = getMeans(sub2);
    sub1.clear();
    sub2.clear();
    inithreshold = (means1 + means2) / 2;
}
long start = System.currentTimeMillis();
System.out.println("Final threshold = " + finalthreshold);
long endTime = System.currentTimeMillis() - start;
System.out.println("Time consumes : " + endTime);
return finalthreshold;
```

完整二值化灰度图像的代码如下，可以通过参数设置来选择哪种方法实现二值化。

```
package com.book.chapter.six;

import java.awt.image.BufferedImage;
import java.util.ArrayList;
import java.util.List;

import com.book.chapter.four.AbstractBufferedImageOp;

public class BinaryFilter extends AbstractBufferedImageOp {
    public final static int MEAN_THRESHOLD = 2;
    public final static int SHIFT_THRESHOLD = 4;

    private int thresholdType;
```

```

public BinaryFilter() {
    thresholdType = MEAN_THRESHOLD;
}

public int getThresholdType() {
    return thresholdType;
}

public void setThresholdType(int thresholdType) {
    this.thresholdType = thresholdType;
}

@Override
public BufferedImage filter(BufferedImage src, BufferedImage dest) {
    int width = src.getWidth();
    int height = src.getHeight();
    if (dest == null)
        dest = createCompatibleDestImage(src, null);

    int[] inPixels = new int[width * height];
    int[] outPixels = new int[width * height];
    getRGB(src, 0, 0, width, height, inPixels);
    int index = 0;
    int means = (int) getThreshold(inPixels, height, width);
    for (int row = 0; row < height; row++) {
        int ta = 0, tr = 0, tg = 0, tb = 0;
        for (int col = 0; col < width; col++) {
            index = row * width + col;
            ta = (inPixels[index] >> 24) & 0xff;
            tr = (inPixels[index] >> 16) & 0xff;
            tg = (inPixels[index] >> 8) & 0xff;
            tb = inPixels[index] & 0xff;
            if (tr > means) {
                tr = tg = tb = 255; // white
            } else {
                tr = tg = tb = 0; // black
            }
            outPixels[index] = (ta << 24) | (tr << 16) | (tg << 8) | tb;
        }
    }
    setRGB(dest, 0, 0, width, height, outPixels);
    return dest;
}

private double getThreshold(int[] pixels, int width, int height) {
    int index = 0;
    double mean = 0;
    if (thresholdType == MEAN_THRESHOLD) {
        // calculate mean, MAX, MIN
        int max = 0;

```

```

int min = 255;
double sum = 0.0;
for (int row = 0; row < height; row++) {
    int tr = 0;
    for (int col = 0; col < width; col++) {
        index = row * width + col;
        tr = (pixels[index] >> 16) & 0xff;
        min = Math.min(min, tr);
        max = Math.max(max, tr);
        sum += tr;
    }
    mean = sum / (width * height);
} else if (thresholdType == SHIFT_THRESHOLD) {
    mean = getMeanShiftThreshold(pixels, height, width);
}
return mean;
}

private static int getMeans(List<Integer> data) {
    int result = 0;
    int size = data.size();
    for (Integer i : data) {
        result += i;
    }
    return (result / size);
}

private int getMeanShiftThreshold(int[] inPixels, int height, int width) {
    // maybe this value can reduce the calculation consume;
    int inithreshold = 127;
    int finalthreshold = 0;
    int temp[] = new int[inPixels.length];
    for (int index = 0; index < inPixels.length; index++) {
        temp[index] = (inPixels[index] >> 16) & 0xff;
    }
    List<Integer> sub1 = new ArrayList<Integer>();
    List<Integer> sub2 = new ArrayList<Integer>();
    int means1 = 0, means2 = 0;
    while (finalthreshold != inithreshold) {
        finalthreshold = inithreshold;
        for (int i = 0; i < temp.length; i++) {
            if (temp[i] <= inithreshold) {
                sub1.add(temp[i]);
            } else {
                sub2.add(temp[i]);
            }
        }
        means1 = getMeans(sub1);
        means2 = getMeans(sub2);
        sub1.clear();
    }
}

```

```

        sub2.clear();
        inithreshold = (means1 + means2) / 2;
    }
    long start = System.currentTimeMillis();
    System.out.println("Final threshold = " + finalthreshold);
    long endTime = System.currentTimeMillis() - start;
    System.out.println("Time consumes : " + endTime);
    return finalthreshold;
}
}

```

测试 BinaryFilter 类时，同样只需要在第3章中提到的 ImagePanel 的 process 方法中添加如下代码，然后在 Eclipse 中运行 MainUI.java 即可。

```

BinaryFilter filter = new BinaryFilter();
destImage = filter.filter(sourceImage, null);

```

选择一张灰度图像，然后单击【处理】按钮即可查看效果。

6.3 图像直方图

图像直方图是图像处理中最重要的概念之一，在各种关于图像处理的书籍中都可以看到它的身影。在介绍直方图概念之前，请看一下图 6-1。

在图 6-1 中，X 轴坐标 0 ~ 255 代表像素值的取值范围，Y 轴则表示 0 ~ 255 之间的各个整数在像素数组中的出现频率。假设一幅图像的像素数组 $P = \{233, 2, 4, 4, 233, 4, 4\}$ ，不难统计出值为 4 的像素出现频率为 4，值为 233 的像素出现频率为 2，值为 2 的像素出现频率为 1。通过这样的简单计算，即可得到

0 ~ 255 各个值在像素数组出现的频率。然后就可以根据像素值与出现频率之间的对应关系生成直方图，即图像像素直方图。计算直方图的数学公式如下：

$$P(g) = \frac{h(g)}{M}$$

其中 $M = \text{image_width} \times \text{image_height}$ ， $h(g)$ 表示灰度为 g 的像素出现频率， g 的取值范围为 0 ~ 255 之间。所以直方图的数学公式还可以为：

$$\text{Histogram} = \sum_{g=0}^{255} \frac{h(g)}{M} = \frac{M}{M} = 1$$

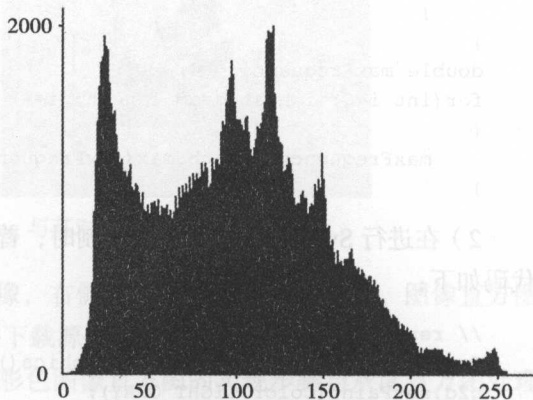


图 6-1 灰度直方图

直方图的意义在于可以通过它直观地观察到图像的对比度、亮度，通过直方图均衡化、直方图匹配也可实现对图像的调整，得到各种想要的处理结果。在学习了直方图相关简单数学知识以后，我们一起学习如何通过编程实现图像直方图。这里使用的图像是灰度图像，RGB 图像的处理与其类似，只是在三个分量上分别绘制出对应的三个直方图。

编程绘制直方图首先要获取图像的直方图数据，然后通过 Swing 2D 实现直方图的绘制与显示，所以整个过程大致可以分为如下两步。

1) 统计图像像素值出现的频率，获取直方图数据，实现代码如下：

```
// get histogram data
int[] histogram = new int[256];
for(int i=0; i<histogram.length; i++)
{
    histogram[i] = 0;
}
for (int row = 0; row < height; row++) {
    int tr = 0;
    for (int col = 0; col < width; col++) {
        index = row * width + col;
        tr = (inPixels[index] >> 16) & 0xff;
        histogram[tr]++;
    }
}
double maxFrequency = 0;
for(int i=0; i<histogram.length; i++)
{
    maxFrequency = Math.max(maxFrequency, histogram[i]);
}
```

2) 在进行 Swing 2D 直方图的绘制时，首先绘制 XY 轴，然后根据直方图数据绘制图形，代码如下：

```
// render the histogram graphic
Graphics2D g2d = dest.createGraphics();
g2d.setPaint(Color.LIGHT_GRAY);
g2d.fillRect(0, 0, width, height);

// draw XY Axis
g2d.setPaint(Color.BLACK);
g2d.drawLine(50, 50, 50, height - 50);
g2d.drawLine(50, height-50, width-50, height - 50);
// draw XY Title
g2d.drawString("0", 50, height-30);
g2d.drawString("255", width-50, height-30);
g2d.drawString("0", 20, height-50);
g2d.drawString("" + maxFrequency, 20, 50);
// draw histogram bar
double xunit = (width - 100.0)/256.0d;
double yunit = (height - 100.0)/maxFrequency;
```

```

for(int i=0; i<histogram.length; i++)
{
    double xp = 50 + xunit * i;
    double yp = yunit * histogram[i];
    Rectangle2D rect2d = new Rectangle2D.
        Double(xp, height - 50 - yp, xunit, yp);
    g2d.fill(rect2d);
}

```

运行与测试图像直方图程序同样只需要在第3章中提到的 ImagePanel 的 process 方法中添加如下代码即可：

```

HistogramFilter filter = new HistogramFilter ();
destImage = filter.filter(sourceImage, null);

```

程序运行的结果图像如图 6-2 所示。

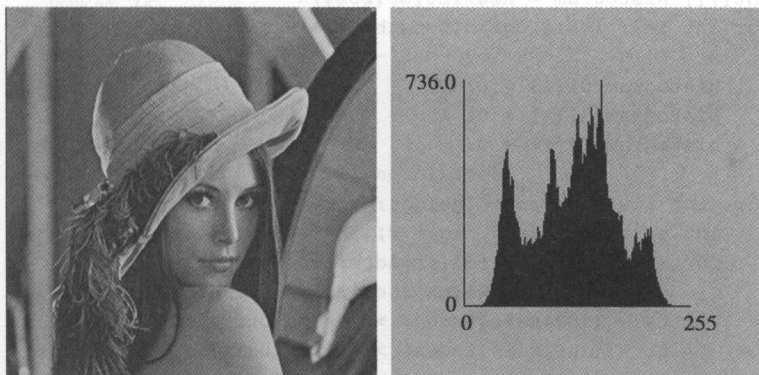


图 6-2 图像 lena_gray.png 与其对应灰度直方图

对于上面的运行结果，左侧为原始输入图像，右侧为该图像对应的直方图。图像直方图完整的源程序清单 HistogramFilter.java 参见本书下载源代码的对应章节。

上面讲述了灰度图像的直方图实现，RGB 彩色图像直方图的实现步骤与灰度直方图处理大致相同，只是要在 RGB 三个颜色通道上进行处理，实现的代码如下：

```

package com.book.chapter.six;

import java.awt.Color;
import java.awt.Graphics2D;
import java.awt.geom.Rectangle2D;
import java.awt.image.BufferedImage;

import com.book.chapter.four.AbstractBufferedImageOp;

public class RGBHistogramFilter extends AbstractBufferedImageOp {

    public RGBHistogramFilter()

```

```

{
    System.out.println("Colorful Histogram");
}

@Override
public BufferedImage filter(BufferedImage src, BufferedImage dest) {
    int width = src.getWidth();
    int height = src.getHeight();

    if (dest == null)
        dest = createCompatibleDestImage(src, null);

    int[] inPixels = new int[width * height];
    getRGB(src, 0, 0, width, height, inPixels);
    int index = 0;
    // get histogram data
    int[][] histogram = new int[3][256];
    for(int i=0; i<histogram.length; i++)
    {
        histogram[0][i] = 0;
        histogram[1][i] = 0;
        histogram[2][i] = 0;
    }
    for (int row = 0; row < height; row++) {
        int ta = 0, tr = 0, tg = 0, tb = 0;
        for (int col = 0; col < width; col++) {
            index = row * width + col;
            ta = (inPixels[index] >> 24) & 0xff;
            tr = (inPixels[index] >> 16) & 0xff;
            tg = (inPixels[index] >> 8) & 0xff;
            tb = inPixels[index] & 0xff;
            histogram[0][tr]++; // red
            histogram[1][tg]++; // green
            histogram[2][tb]++; // blue
        }
    }
    double[] maxRGBFrequency = new double[]{0,0,0};
    for(int i=0; i<histogram[0].length; i++)
    {
        maxRGBFrequency[0] = Math.max(maxRGBFrequency[0], histogram[0][i]);
        maxRGBFrequency[1] = Math.max(maxRGBFrequency[1], histogram[1][i]);
        maxRGBFrequency[2] = Math.max(maxRGBFrequency[2], histogram[2][i]);
    }

    // render the histogram graphic
    Graphics2D g2d = dest.createGraphics();
    g2d.setPaint(Color.LIGHT_GRAY);
    g2d.fillRect(0, 0, width, height);
    double max = Math.max(maxRGBFrequency[2],
        Math.max(maxRGBFrequency[0], maxRGBFrequency[1]));

```

```

// draw XY Axis
g2d.setPaint(Color.BLACK);
g2d.drawLine(50, 50, height - 50);
g2d.drawLine(50, height-50, width-50, height - 50);
// draw XY Title
g2d.drawString("0", 50, height-30);
g2d.drawString("255", width-50, height-30);
g2d.drawString("0", 20, height-50);
g2d.drawString("" + max, 20, height-50);
// draw histogram bar
double xunit = (width - 100.0)/256.0d;
double yunit = (height - 100.0)/max;
g2d.setPaint(Color.RED);
for(int i=0; i<histogram[0].length; i++)
{
    double xp = 50 + xunit * i;
    double yp = yunit * histogram[0][i];
    Rectangle2D rect2d = new Rectangle2D.Double(xp, height - 50 - yp, xunit, yp);
    g2d.fill(rect2d);
}
g2d.setPaint(Color.GREEN);
for(int i=0; i<histogram[1].length; i++)
{
    double xp = 50 + xunit * i;
    double yp = yunit * histogram[1][i];
    Rectangle2D rect2d = new Rectangle2D.Double(xp, height - 50 - yp, xunit, yp);
    g2d.fill(rect2d);
}
g2d.setPaint(Color.BLUE);
for(int i=0; i<histogram[2].length; i++)
{
    double xp = 50 + xunit * i;
    double yp = yunit * histogram[2][i];
    Rectangle2D rect2d = new Rectangle2D.Double(xp, height - 50 - yp, xunit, yp);
    g2d.fill(rect2d);
}
return dest;
}
}

```

测试彩色图像只要修改 ImagePanel.java 中的 process 方法, 添加如下代码即可:

```

RGBHistogramFilter filter = new RGBHistogramFilter();
destImage = filter.filter(sourceImage, null);

```

然后在 Eclipse 中运行 MainUI.java。

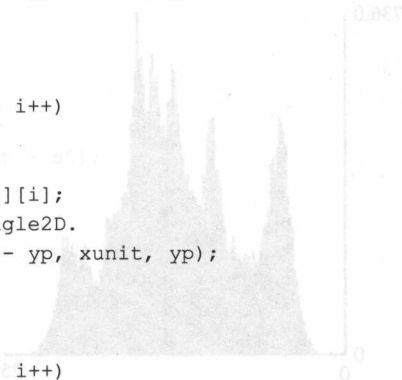


图 6-3 彩色直方图

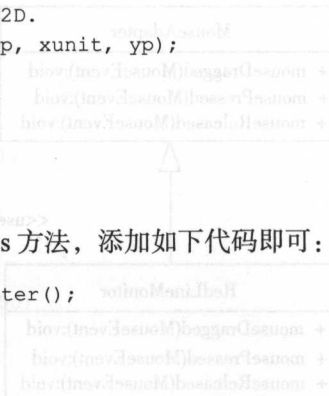


图 6-4 类图

6.4 基于直方图实现图像二值化

本节将学习通过直方图的数据寻找合适的阈值，并实现灰度图像的二值化。首先看一下下面的直方图（如图 6-3 所示，其对应的灰度图像为 lena_gray.png）。

观察图 6-3 不难发现，有两个比较明显的波谷，一个靠近 0，一个靠近 255。所以从直方图上可以判断合适的二值化阈值应该在这两个波谷之间。假设第一个波谷的值为 $T1$ ，第二个波谷的值为 $T2$ ，则大致计算阈值的公式为 $T = \frac{T1 + T2}{2}$ 。但是当图像直方图为如图 6-4 所示的结果时，显然，此时合适的二值化图像阈值应该是在波峰位置左右，所以只要在直方图上通过 Java 2D 绘制出一条可以支持鼠标拖动的直线，实时显示阈值与图像二值化以后的结果，通过不断尝试就可以得到想要的阈值与二值图像。显然，这样的程序在多数的图像处理与编辑软件中非常常见，下面就一起通过编码来实现该程序。首先看看实现的各个类之间的关系图，如图 6-5 所示。

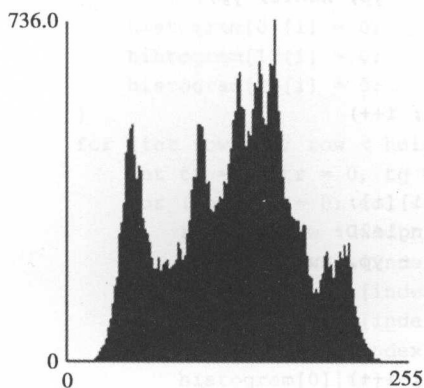


图 6-3 lena_gray.png 对应的灰度直方图

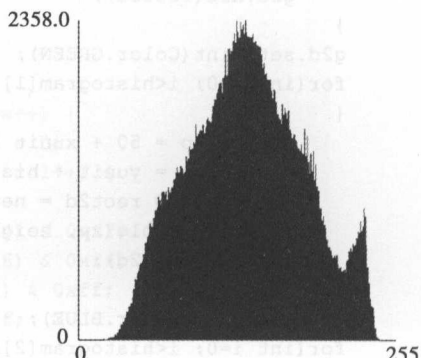


图 6-4 直方图二

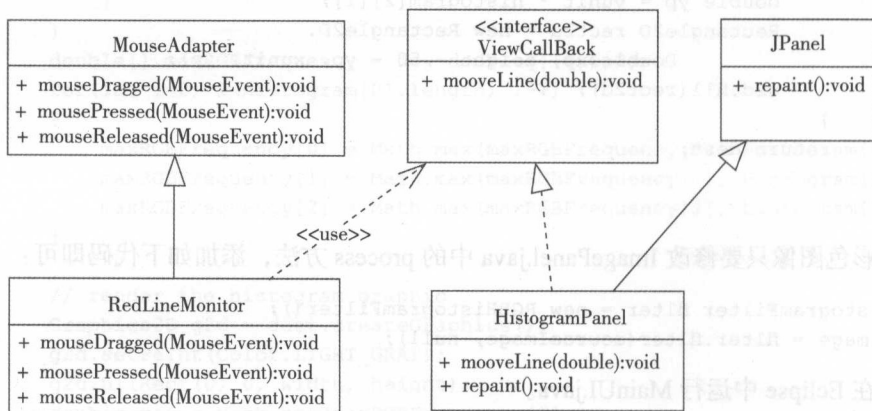


图 6-5 各个类之间的关系图

其中

❑ RedLineMonitor 类：实现鼠标移动位置的捕获。

❑ HistogramPanel 类：实现直方图的显示与阈值直线移动，根据鼠标位置计算出直方图上的阈值。

实现直方图绘制与阈值直线移动的代码如下：

```
int width = (int)size.getWidth();
int height = (int)size.getHeight();
double maxFrequency = 0;
for(int i=0; i<data.length; i++)
{
    maxFrequency = Math.max(maxFrequency, data[i]);
}

// render the histogram graphic
Graphics2D g2d = histogramImage2.createGraphics();
g2d.setPaint(Color.LIGHT_GRAY);
g2d.fillRect(0, 0, width, height);

// draw XY Axis
g2d.setPaint(Color.BLACK);
g2d.drawLine(50, 50, 50, height - 50);
g2d.drawLine(50, height-50, width-50, height - 50);

// draw XY Title
g2d.drawString("0", 50, height-30);
g2d.drawString("255", width-50, height-30);
g2d.drawString("0", 20, height-50);
g2d.drawString("" + maxFrequency, 20, 50);

// draw histogram bar
double xunit = (width - 100.0)/256.0d;
double yunit = (height - 100.0)/maxFrequency;
for(int i=0; i<data.length; i++)
{
    double xp = 50 + xunit * i;
    double yp = yunit * data[i];
    Rectangle2D rect2d = new Rectangle2D.Double(xp, height - 50 - yp, xunit, yp);
    g2d.fill(rect2d);
}

// render red line
if((linePos - 50) >= 0 && (width - linePos)>= 50)
{
    threshold = (int)((linePos - 50) / xunit);
    linePos = 50 + xunit * threshold;
    g2d.setPaint(Color.RED);
    g2d.drawLine((int)linePos, 50,
```



```

        (int)linePos, height - 50);
    g2d.drawString("阈值:"+threshold,
        (int)linePos-10, 50);
}

```

❑ **HistogramDataExtractor** 类：用来提取输入图像的直方图数据，然后根据直方图数据选择阈值实现灰度图像二值化。实现的代码如下：

```

package com.book.chapter.six;

import java.awt.image.BufferedImage;
import com.book.chapter.four.AbstractBufferedImageOp;

public class HistogramDataExtractor extends
    AbstractBufferedImageOp {
    private int threshold = -1;

    public void setThreshold(int threshold) {
        this.threshold = threshold;
    }

    private int[] histogram;

    public int[] getHistogram() {
        return histogram;
    }

    public void setHistogram(int[] histogram) {
        this.histogram = histogram;
    }

    @Override
    public BufferedImage filter(BufferedImage src,
        BufferedImage dest) {
        int width = src.getWidth();
        int height = src.getHeight();

        if (dest == null)
            dest = createCompatibleDestImage(src, null);

        int[] inPixels = new int[width * height];
        getRGB(src, 0, 0, width, height, inPixels);
        int index = 0;

        // get histogram data
        histogram = new int[256];
        for(int i=0; i<histogram.length; i++)
        {
            histogram[i] = 0;
        }
        for (int row = 0; row < height; row++) {

```

```

    int tr = 0;
    for (int col = 0; col < width; col++) {
        index = row * width + col;
        tr = (inPixels[index] >> 16) & 0xff;
        histogram[tr]++;
    }
}

if(threshold > 0)
{
    // binary image
    int[] outPixels = new int[width*height];
    for(int row=0; row<height; row++) {
        int ta = 0, tr = 0, tg = 0, tb = 0;
        for(int col=0; col<width; col++) {
            index = row * width + col;
            ta = (inPixels[index] >> 24) & 0xff;
            tr = (inPixels[index] >> 16) & 0xff;
            tg = (inPixels[index] >> 8) & 0xff;
            tb = inPixels[index] & 0xff;
            if(tr >=threshold) {
                tr = tg = tb = 255;
            } else {
                tr = tg = tb = 0;
            }
            outPixels[index] = (ta << 24) | (tr << 16)
                               | (tg << 8) | tb;
        }
        setRGB( dest, 0, 0, width, height, outPixels );
    }
    return dest;
}
}

```

上述三个类实现以后，只需要在 ImagePanel 的 process 方法中添加如下代码即可调用：

```

final HistogramDataExtractor extractor =
    new HistogramDataExtractor();
destImage = extractor.filter(sourceImage, null);
int[] histData = extractor.getHistogram();
final HistogramPanel uiPanel = new
    HistogramPanel(destImage, histData);
RedLineMonitor lineListener = new RedLineMonitor(uiPanel);
uiPanel.addMouseListener(lineListener);
uiPanel.addMouseMotionListener(lineListener);
uiPanel.addActionListener(new ActionListener(){

    @Override
    public void actionPerformed(ActionEvent e) {

```

```

        extractor.setThreshold(uiPanel.getThreshold());
        destImage = extractor.filter(sourceImage, null);
        refresh();
    }

    //HistogramDataExtractor 类：用来提取输入图像的直方图数据，然后根据直方图数据选
    //择二值化。实现的代码如下。
    uiPanel.openView();

```

然后运行 MainUI.java 即可查看整个程序的运行效果，如图 6-6 所示。



图 6-6 直方图二值化运行结果

在图 6-6 中，最左边的是输入图片 lena_gray.png，右边的是二值化以后的图像，对话框中是直方图图像，其中的直线，鼠标拖动时会移动并且改变显示值。本节完整的源代码文件请下载阅览。

6.5 应用——直方图均衡化

本节重点介绍一种修改直方图分布的算法——直方图均衡化，通过该方法实现对图像的增强与提高，获得更好的图像修复效果，首先来看一下直方图均衡化的数学表达与理论计算。假设：

输入图像的灰度级别为 $r \in [0, 1]$ ，变换以后图像的灰度级别为 $s \in [0, 1]$ ，则可以得到 $s = T(r)$ ，其中 $T(r)$ 为变换公式。

对于一幅离散的数字图像来说，均衡化变换公式可以表示为：

$$S_k = T(r_k) = \sum_{j=0}^k Pr(r_j) = \sum_{j=0}^k \frac{n_j}{n}$$

其中 $Pr(r_j)$ 是图像直方图公式。

下面介绍一下灰度图像直方图均衡化步骤：

1) 计算输入图像的直方图统计。

2) 根据直方图均衡化公式计算变换后的直方图。

3) 重新映射到像素值, 实现图像调整。

下面是彩色 RGB 图像直方图均衡化步骤:

1) 将像素从 RGB 转换到 HSI 色彩空间。

2) 计算 I 分量的直方图统计数据。

3) 直方图均衡化。

4) 重新映射 I 值到各个像素。

5) 将像素从 HSI 转换到 RGB 色彩空间。

从上面可以观察到, 灰度图像与彩色图像的直方图均衡化唯一不同的是, 彩色图像需要将像素值从 RGB 色彩空间转换到 HSI 色彩空间, 之后的处理与灰度图像类似。最后还需要将像素值从 HSI 转换到 RGB 色彩空间, 即可得到最终效果。

下面就来具体学习 RGB 色彩空间与 HSI 色彩空间相互转换的知识, RGB 到 HSI 色彩空间转换遵循以下步骤:

1) 归一化像素值 $r = \frac{R}{R+G+B}$, $g = \frac{G}{R+G+B}$, $b = \frac{B}{R+G+B}$

2) 根据 r 、 g 、 b 值获得归一化 HSI 值。

$$h = \cos^{-1} \frac{0.5 \times [(r-g) + (r-b)]}{[(r-g)^2 + (r-b)(g-b)]^{1/2}} \quad h \in [0, \pi] \text{ 当 } b \leq g \text{ 时}$$

$$h = 2\pi - \cos^{-1} \frac{0.5 \times [(r-g) + (r-b)]}{[(r-g)^2 + (r-b)(g-b)]^{1/2}} \quad h \in [\pi, 2\pi] \text{ 当 } b > g \text{ 时}$$

$$s = 1 - 3 \times \min(r, g, b), \quad s \in [0, 1]$$

$$i = \frac{R+G+B}{3 \times 255}, \quad i \in [0, 1]$$

3) 根据 HSI 的取值范围分别为 $H \in [0, 360]$ 、 $S \in [0, 100]$ 、 $I \in [0, 255]$ 得到

$$H = h \times 180/\pi; \quad S = s \times 100; \quad I = i \times 255$$

从 HSI 色彩空间到 RGB 色彩空间则需要如下计算步骤方可完成:

1) 首先归一化 HSI 值: $h = H \times \frac{\pi}{180}$, $s = S/100$, $i = I/255$

2) 计算中间过程三个分量值 x 、 y 、 z :

$$x = i \times (1 - s), y = i \times \left[1 + \frac{s \times \cos(h)}{\cos\left(\frac{\pi}{3} - h\right)} \right], z = 3i - (x + y)$$

3) 根据 h 的范围

当 $h < \frac{2\pi}{3}$ 时, $b = x$, $r = y$, $g = z$

当 $\frac{2\pi}{3} \leq h < \frac{4\pi}{3}$ 时, $h = h - \frac{2\pi}{3}$, $r = x$, $g = y$, $b = z$

当 $\frac{4\pi}{3} \leq h < 2\pi$ 时, $h = h - \frac{4\pi}{3}$, $g = x$, $b = y$, $r = z$

上述 RGB 与 HSI 色彩空间相互转换的方法为几何推导法, 至于公式是如何推导出来的, 感兴趣的读者可以自己研究。这里主要是学会根据公式实现应用, 解决实际问题。首先看一下实现 RGB 与 HSI 色彩空间相互转换的代码。

RGB 转换到 HSI 色彩空间的代码如下:

```
double[] rgb2HSI(int[] rgb)
{
    double sum = rgb[0] + rgb[1] + rgb[2];
    double r = rgb[0] / sum;
    double g = rgb[1] / sum;
    double b = rgb[2] / sum;
    double s1 = ((r-g) + (r-b))/2.0f;
    double s2 = Math.pow((r-g), 2) + (r-b)*(g-b);
    double s3 = s1/Math.sqrt(s2);
    double h = 0.0f;
    if (b<=g)
    {
        h = Math.acos(s3);
    }
    else if (b>g)
    {
        h = 2*Math.PI - Math.acos(s3);
    }
    double s = 1 - 3*Math.min(r, Math.min(g, b));
    double i = sum / (255.0 * 3.0);

    // 得到输出值
    double H = (h*180.0)/Math.PI;
    double S = s * 100.0;
    double I = i * 255.0;

    return new double[]{H, S, I};
}
```

HSI 转换到 RGB 色彩空间的代码如下:

```
int[] hsi2RGB(double[] hsi)
{
    double h = (hsi[0] * Math.PI)/180.0;
    double s = hsi[1] / 100.0;
    double i = hsi[2] / 255.0;
    double x = i*(1-s);
```

```

double y = i*(1 + (s*Math.cos(h))/Math.cos(Math.PI/3.0 - h));
double z = 3*i - (x + y);
double r=0, g=0, b=0;
if(h < ((2*Math.PI)/3))
{
    b = x;
    r = y;
    g = z;
}
else if(h >= ((2*Math.PI)/3) && h < ((4*Math.PI)/3))
{
    h = h - ((2*Math.PI)/3.0);
    x = i*(1-s);
    y = i*(1 + (s*Math.cos(h))/Math.cos(Math.PI/3.0 - h));
    z = 3*i - (x + y);
    r = x;
    g = y;
    b = z;
}
else if(h >= ((4*Math.PI)/3) && h < ((2*Math.PI)))
{
    h = h - ((4*Math.PI)/3.0);
    x = i*(1-s);
    y = i*(1 + (s*Math.cos(h))/Math.cos(Math.PI/3.0 - h));
    z = 3*i - (x + y);
    g = x;
    b = y;
    r = z;
}
int red = (int)(r * 255);
int green = (int)(g * 255);
int blue = (int)(b * 255);

return new int[]{red, green, blue};
}

```

解决了像素色彩空间转换问题之后,彩色图像的直方图均衡化就很容易通过编码实现了。

下面是实现彩色图像直方图均衡化 HistogramEFilter 类的关键代码:

```

public BufferedImage filter(BufferedImage src, BufferedImage dest) {
    int width = src.getWidth();
    int height = src.getHeight();

    if (dest == null)
        dest = createCompatibleDestImage(src, null);

    int[] inPixels = new int[width*height];
    double[][] hsiPixels = new double[3][width*height];
    int[] outPixels = new int[width*height];
    getRGB(src, 0, 0, width, height, inPixels);
}

```



```

int[] iDataBins = new int[256]; // RGB
int[] newiBins = new int[256]; // after HE
for(int j=0; j<256; j++) {
    iDataBins[j] = 0;
    newiBins[j] = 0;
}

int index = 0;
int totalPixelNumber = height * width;
for(int row=0; row<height; row++) {
    int ta = 0, tr = 0, tg = 0, tb = 0;
    for(int col=0; col<width; col++) {
        index = row * width + col;
        ta = (inPixels[index] >> 24) & 0xff;
        tr = (inPixels[index] >> 16) & 0xff;
        tg = (inPixels[index] >> 8) & 0xff;
        tb = inPixels[index] & 0xff;

        double[] hsi = rgb2HSI(new int[]{tr, tg, tb});
        iDataBins[(int)hsi[2]]++;
        hsiPixels[0][index] = hsi[0];
        hsiPixels[1][index] = hsi[1];
        hsiPixels[2][index] = hsi[2];
    }
}

// generate original source image RGB histogram
generateHEData(newiBins, iDataBins, totalPixelNumber, 256);
for(int row=0; row<height; row++) {
    int ta = 255, tr = 0, tg = 0, tb = 0;
    for(int col=0; col<width; col++) {
        index = row * width + col;

        // get output pixel now...
        double h = hsiPixels[0][index];
        double s = hsiPixels[1][index];
        double i = newiBins[
            (int)hsiPixels[2][index]];
        int[] rgb = hsi2RGB(new double[]{h, s, i});
        tr = clamp(rgb[0]);
        tg = clamp(rgb[1]);
        tb = clamp(rgb[2]);
        outPixels[index] = (ta << 24)
            | (tr << 16) | (tg << 8) | tb;
    }
}

setRGB( dest, 0, 0, width, height, outPixels );
return dest;
}

```

完整的 HistogramEFilter 类代码请从前言中所述网站下载阅览。测试直方图均衡化时，只

需要在 ImagePanel 的 process 方法中添加如下代码:

```
HistogramEFilter heFilter = new HistogramEFilter();
destImage = heFilter.filter(sourceImage, null);
```

然后运行 MainUI.java 即可查看效果, 本节的测试图片为 lena.png。



注意 如果没有特殊说明, 本书所用到的灰度图片资源均为 lena_gray.png, 彩色图片均为 lena.png, 当然读者也可以选择自己想要的图片来测试程序效果。

6.6 应用——基于直方图的图像搜索

本节将基于直方图做图像相似度的比较, 实现以图搜图的应用, 这在实际项目中是一种很实用的技术。当前实现图像匹配的算法多数是基于图像特征查找的, 首先提取图像特征, 然后根据特征值进行分析与计算, 从而实现图像匹配。最常见的是基于 sift 的算法实现, 但是计算量较大。这里介绍一种基于图像直方图实现的图像匹配算法, 优点是计算量小, 相比 sift 算法, 原理简单, 只需要少许数学知识即可。

1. 基本原理

通过分析比较两幅图像的直方图数据, 从而确定这两幅图像的相似度, 首先获取图像的直方图数据, 接着计算两个直方图数据之间的差异, 可以选择的计算两组数据差异与距离的公式有巴氏距离、欧几里得距离、地球移动距离等, 最终得到一个相似度输出值。

2. 实现图像相似度分析步骤

具体步骤如下:

- 1) 提取输入图像的直方图数据。
- 2) 根据输入图像的直方图数据, 运用距离公式对每一个待比较图像实现相似度计算。
- 3) 显示图像的对比度信息。

3. 程序难点剖析: 彩色图像的直方图表示

彩色图像 RGB 拥有三个色彩分量, 每个分量都可以表示为一个直方图, 所以首先要做的就是如何用一个直方图表示 RGB 三个分量颜色值的变化, 通常的做法就是将 RGB 颜色值的取值范围进行分段表示, 将 0 到 255 的颜色值分为 16 个灰度级别, 每个灰度级别拥有 16 个灰度值, 这样 RGB 直方图数据就可以通过 $16 \times 16 \times 16$ 的一维数组表示出来了。对于给定的任意像素 $P(r, g, b)$, 其对应的直方图数据索引为:

$$index = (r/16) + (g/16) \times 16 + (b/16) \times 16 \times 16$$

任意的直方图数据 $histogram[index]++$, 对于给定的像素 $P(r, g, b)$, 每个通道通过除以 16 取整的值范围为 $[0 \sim 15]$, 所以三个通道就得到一个 $16 \times 16 \times 16$ 大小的数组的直方图数据。根据上述原理, 对于任意一幅图像的 RGB 像素值, 可以得到它的直方图数据。实现代码

如下：

```
double[] bins = distanceType == EARTH_MOVERS_DISTANCE
    ? new double[4*4*4] : new double[16*16*16];
int width = image.getWidth();
int height = image.getHeight();
```

```
// 从图像中获取像素数据
int[] inPixels = new int[width * height];
getRGB(image, 0, 0, width, height, inPixels);
int index = 0;
```

```
// 初始化直方图数据
for(int i=0; i<bins.length; i++)
{
    bins[i] = 0;
}
```

```
// 计算RGB每个分量的16 bin的index
for (int row = 0; row < height; row++) {
    int tr = 0, tg=0, tb=0;
    for (int col = 0; col < width; col++) {
        index = row * width + col;
        tr = (inPixels[index] >> 16) & 0xff;
        tg = (inPixels[index] >> 8) & 0xff;
        tb = inPixels[index] & 0xff;
        int level = 16;
        if(distanceType == EARTH_MOVERS_DISTANCE)
        {
            level = 64;
        }
        int rbinIndex = tr / level;
        int gbinIndex = tg / level;
        int bbinIndex = tb / level;
        int binIndex = rbinIndex + gbinIndex * (256/level)
            + bbinIndex * (256/level) * (256/level);
        bins[binIndex]++;
    }
}
```

```
// 归一化直方图数据
float total = width * height;
for(int i=0; i<bins.length; i++)
{
    bins[i] = bins[i] / total;
}
return bins;
```

4. 距离计算

在获取到两张图像的直方图数据以后，就可以进行直方图数据的比较了。从统计学上看，

直方图数据属于统计分布概率,所以可以通过比较两组数据之间的距离完成。在这里为计算方便,假设两张图像的大小一致(对于大小不一致的图像,可以通过 `resize` 之后得到大小一致的图像,然后再进行计算),计算两组数据距离的公式最常见的有欧几里得距离、巴氏系数与地球移动距离。下面分别介绍如何通过这三种距离公式来计算图像直方图的相似度。

(1) 欧几里得距离

$$distance = \sqrt{\sum_{i=0}^N (p_i - q_i)^2}$$

其中 N 表示图像的总像素个数 ($width \times height$),如果两张图像完全一致,则它的 $distance$ 为 0, $distance$ 越大说明图像差别越大。基于欧几里得计算图像直方图相似度的代码如下:

```
double sum = 0;
for(int i=0; i<srcBins.length; i++)
{
    sum += Math.pow((srcBins[i] - destBins[i]), 2);
}
return Math.sqrt(sum);
```

(2) 巴氏系数

$$\rho = \sum_{u=1}^m \sqrt{(p_u q_u)}$$

其中 p_u 与 q_u 分别表示两张图像的直方图数据,计算出来的值在 $0 \sim 1$ 之间,1 表示完全相同,0 表示完全不相同。基于巴氏系数比较图像直方图相似度的代码如下:

```
double[] mixedData = new double[srcBins.length];
for(int i=0; i<srcBins.length; i++) {
    mixedData[i] = Math.sqrt(srcBins[i] * destBins[i]);
}
```

```
// The values of Bhattacharyya Coefficient
// ranges from 0 to 1,
double similarity = 0;
for(int i=0; i<mixedData.length; i++) {
    similarity += mixedData[i];
}
```

```
// The degree of similarity
return similarity;
```

(3) 地球移动距离 (EMD)

Earth Mover's Distance 即两幅图随机分布,其中一个可以看成是地表不规则高低分布的堆,另外一个看成是一系列孔的集合,把第一个堆填到第二个孔所需要的最小距离。

所以从本质上看,EMD 是 `transportation problem` 算法问题,就是一系列商品提供者需要把商品提供给一系列对应的订购者,而它们之间的距离不一样,订购数量也不一样。EMD 的数学公式为:

$$EMD(P, Q) = \frac{\sum_{i=1}^m \sum_{j=1}^n f_{ij} d_{ij}}{\sum_{i=1}^m \sum_{j=1}^n f_{ij}}, \text{ 其中 } \sum_{i=1}^m \sum_{j=1}^n f_{ij} = \min \left(\sum_{i=1}^m W_{p_i}, \sum_{j=1}^n W_{q_j} \right) \text{ 而且 } f_{ij} \geq 0$$

其中, d_{ij} 表示 W_{p_i} 到 W_{q_j} 之间的距离。从上述表达式可以看出, EMD 支持长度可变的数据集比较, 距离计算更加灵活。根据上述数学公式, 实现直方图 EMD 计算的代码如下:

```
Signature sig1 = getSignature(srcBins, 16);
Signature sig2 = getSignature(destBins, 16);
double dist = JFastEMD.distance(sig1, sig2, -1);
return dist;
```

这里特别说明一下, 如何基于 transportation problem 算法实现 EMD 计算是一个很长、很专业的话题, 其中涉及的算法大致可以分为两个部分, 第一步是寻找与计算最小费用, 第二步则为检测计算是否已经是最小费用, 如果不是则继续迭代计算, transportation problem 算法本身是 NP 问题, 更具体讨论已经超出了本节内容范围。这里用到了一个 Java 版 EMD 的算法包, 可以很方便地计算两组直方图数据的 EMD 值, 对 EMD 算法感兴趣的读者可以阅读本书对应章节的相关代码实现。对于两张完全一致的图片, EMD 的计算结果为 0, 两张图像差异越大, EMD 计算结果值越大, 大于 1 时, 基本上可以认为是完全不同的图片。

利用上述三种方法实现图像比较的方法 compareTo, 其代码如下:

```
int width = destImage.getWidth();
int height = destImage.getHeight();
if(width != srcImage.getWidth()
    || height != srcImage.getHeight())
{
    throw new IllegalArgumentException
        ("图像宽度与高度与源图像不符合!");
}
double[] destBins = calculateHistogram(destImage);
if(getDistanceType() == EUCLIDEAN_DISTANCE)
{
    return calculateEuclideanDis(
        getSrcHistogramData(), destBins);
}
else if(getDistanceType() == BHATTACHARYYA_COEFFICIENT)
{
    return calculateBhattacharyya(
        getSrcHistogramData(), destBins);
}
else
{
    return calculateEmd(getSrcHistogramData(), destBins);
}
```


只需对 `HistogramComparisonFilter` 类进行适当的初始化, 然后调用 `compareTo` 方法即可比较源图像与目标图像的相似度, 调用实现的 UI 代码如下:

```
HistogramComparisonFilter heFilter = new
    HistogramComparisonFilter(sourceImage,
        HistogramComparisonFilter.EARTH_MOVERS_DISTANCE);
// File destFile = new File("E:\\image\\test11.png");
File destFile = new File("E:\\image\\lena.png");
try {
    BufferedImage toImage = ImageIO.read(destFile);
    double value = heFilter.compareTo(toImage);
    Graphics2D g2d = toImage.createGraphics();
    g2d.setPaint(Color.RED);
    g2d.drawString("" + value, 50, 50);
    destImage = toImage;
} catch (IOException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
```



注意 特别要提醒的是对 `HistogramComparisonFilter` 中 `compareTo()` 方法返回值的处理, 当利用欧几里得距离与地球移动距离计算相似度时, 值越小, 图像越相似, 0 代表完全相同。而基于巴氏系数则是 1 表示完全相同, 值越小越不同。

本节中提到的其他类与程序源代码请下载阅览, 这里再次特别强调源代码也是本书的一部分, 请认真阅读与对待。

6.7 小结

本章从统计学的基本数学知识入手, 介绍了统计图像像素均值、最大值与最小值、方差等图像像素的统计学属性, 介绍了根据图像像素方差阈值实现过滤空白图像的方法。接着介绍了图像直方图概念与图像直方图获取和显示方法, 以及基于图像直方图寻找图像二值化, 利用直方图实现图像均衡化的原理、方法和代码实现等。最后阐述了利用图像直方图数据实现图像相似度分析比较的三种方法, 完成了读者对图像直方图认知的升华。这些活生生的例子让图像处理理论知识贴近项目应用, 不再显得那么虚无缥缈。关于直方图匹配的话题与 transportation problem 算法问题, 本章限于篇幅未能展开, 强烈建议读者在阅读完本章内容以后进一步去学习, 这有利于强化与巩固本章所学知识。



Chapter 7 第7章

图像编辑

第6章已经学了图像像素统计知识与直方图相关知识，本章将为读者介绍项目最常遇到的图像放缩（resize）、旋转等问题的解决方法。学完本章后，读者应该完全有能力驾驭任何语言实现图像的放缩、旋转等操作，真正实现对图像像素内容的编辑。同样本章内容也会涉及一些最基础的数学知识，但绝对不会太多，而且尽量以直白，程序员可以看得懂的方式阐述这些数学知识，大家在学习本章内容的同时，也可以复习高中的数学。

本章将会从图像像素采样开始，引入图像插值概念，然后介绍几种常见的插值方法，最后介绍图像旋转实现原理与方法，从根本上帮助读者理解、认知这些原理，解决这些非常实际的问题，真正做到授人以渔。

7.1 为什么图像放大以后失真

很多编程语言都支持图像放大与缩小，但很多时候图像放大以后就会很明显地看到边缘有很多小锯齿，这就是图像放大以后失真最直接与有力的证据。当然，也有很多编程语言支持多种不同的放大方法，可以通过参数设置避免放大图像失真。这里要强调的是，图像放大以后失真是指图像明显出现模糊，边缘有锯齿等，而不是细微难以观察到的差别。

从数字图像的本质来说，一个像素可以看成是一个采样（sample），一张图像就是由这许许多多多个采样（像素）组成的。图像分辨率从广义上说可以有三个级别，第一个是像素值的位数，常见的有32位、16位、8位等；第二个是图像空间上宽度与高度；第三是显示器刷新图形图像的频率Hz。

图像放大以后失真的主要原因在于，空间上多出来那些新像素点没有得到正确的采样，

所以图像放大以后失真可以看作图像采样的问题。对于图像 zoom/resize 简单采样的方法可以分为三种, 分别是像素替换, 即 Pixel replacement (临近点插值)、零阶保持采样、K 阶采样 (可以对图像任意 zoom)。下面就来学习这三种放大图像采样方法。

1. 像素替换放大

像素替换放大 (Pixel replacement zooming) 与一种常见的图像放缩方法临近点插值相似, 本章稍后会进一步介绍临近点插值算法, 这里首先介绍像素替换, 其对图像的放大 (zoom) 只是取最相邻像素点像素值作为放大后的图像像素 (采样)。工作原理也很简单, 如果图像放大 n 倍, 每个像素都会被替换 n 次。假设图像宽与高放大 2 倍, 原来像素值为 1、2、3、4, 图像宽与高都为 2, 表示如下:

1	2
3	4

原像素矩阵

X 轴方向的像素替换放大 2 倍以后的像素矩阵表示如下:

1	1	2	2
3	3	4	4

Y 轴方向的像素替换放大 2 倍以后的像素矩阵如下:

1	1	2	2
1	1	2	2
3	3	4	4
3	3	4	4

可以看出最终图像放大为原来的 4 倍了。像素替换实现图像放大的代码如下:

```
int nrow = (int)(row / times);
int ncol = (int)(col / times);
int index2 = nrow * width + ncol;
index = row * width + col;
outPixels[index] = inPixels[index2];
```

其中 inPixels 表示输入图片的像素数组, outPixels 表示放大以后的像素数组。

2. 零阶保持放大

零阶保持放大 (Zero-order hold zooming) 图片也被称为双倍放大, 是通过间隔插值实现对图像放大的方法, 所以这种放大图像的方法适合对图像实现 2^n 整数倍放大。相对第一种方法, 该方法不会产生模糊放大效果, 它的基本思想如下: 选择两个相邻的像素值相加再除以 2, 然后将此值插入两个像素之间作为新的像素值, 在对每一行与每一列的间隔插值以后, 图像的

宽和高就各自放大了两倍。假设图像像素数组 2×2 ，宽高均为 2，表示如下：

1	2
3	4

对第一行进行零阶保持采样， $(1+2)/2 = 1.5$ 取整为 1，对第二行进行同样计算取整为 3，所以得到的结果如下：

1	1	2
3	3	4

对每一列进行零阶保持技术，第一列 $(1+3)/2 = 2$ ，第二列 $(1+3)/2 = 2$ ，第三列 $(2+4)/2 = 3$ ，所以采样放大得到最终结果如下：

1	1	2
2	2	3
3	3	4

通过该方法实现采样放大的代码如下：

```
// for each row
int index = 0;
int[] rowPixels = new int[dw * height];
for (int row = 0; row < height; row++) {
    int ta = 0, tr = 0, tg = 0, tb = 0;
    for (int col = 0; col < width; col++) {
        index = row * width + col;
        ta = (inPixels[index] >> 24) & 0xff;
        tr = (inPixels[index] >> 16) & 0xff;
        tg = (inPixels[index] >> 8) & 0xff;
        tb = inPixels[index] & 0xff;
        int pcol = col - 1;
        if (pcol < 0) {
            pcol = 0;
        }
        int index2 = row * width + pcol;
        int ta2 = (inPixels[index2] >> 24) & 0xff;
        int tr2 = (inPixels[index2] >> 16) & 0xff;
        int tg2 = (inPixels[index2] >> 8) & 0xff;
        int tb2 = inPixels[index2] & 0xff;
        int tr3 = (tr + tr2) / 2;
        int tg3 = (tg + tg2) / 2;
        int tb3 = (tb + tb2) / 2;
        int ncol = col * 2 - 1;
        if (ncol < 0)
```

```

{
    ncol = 0;
}
index = row * dw + ncol;
rowPixels[index] = (ta << 24)
    | (tr3 << 16) | (tg3 << 8) | tb3;
index = row * dw + col * 2;
rowPixels[index] = (ta << 24)
    | (tr << 16) | (tg << 8) | tb;
}

// for each column
for (int col = 0; col < dw; col++) {
    int ta = 0, tr = 0, tg = 0, tb = 0;
    for (int row = 0; row < height; row++) {
        index = row * dw + col;
        ta = (rowPixels[index] >> 24) & 0xff;
        tr = (rowPixels[index] >> 16) & 0xff;
        tg = (rowPixels[index] >> 8) & 0xff;
        tb = rowPixels[index] & 0xff;
        int prow = (row - 1) < 0 ? 0 : (row - 1);
        int index2 = prow * dw + col;
        int ta2 = (rowPixels[index2] >> 24) & 0xff;
        int tr2 = (rowPixels[index2] >> 16) & 0xff;
        int tg2 = (rowPixels[index2] >> 8) & 0xff;
        int tb2 = rowPixels[index2] & 0xff;
        int tr3 = (tr + tr2)/2;
        int tg3 = (tg + tg2)/2;
        int tb3 = (tb + tb2)/2;
        int nrow = row*2 - 1;
        if(nrow < 0)
        {
            nrow = 0;
        }
        index = nrow * dw + col;
        outPixels[index] = (ta << 24)
            | (tr3 << 16) | (tg3 << 8) | tb3;
        index = (row * 2) * dw + col;
        outPixels[index] = (ta << 24)
            | (tr << 16) | (tg << 8) | tb;
    }
}

```

其中 inPixels 表示输入图像数组，outPixels 表示输出图像数组。rowPixels 用来存储行变换以后的像素数组。其实零阶保持采样放大本质上是双线性插值的变种，关于双线性插值，后面章节将详细介绍，这里不再表述。

3. K 次放大

相比前面两种方法，K 次放大 (K-times zooming) 更值得关注，因为 K 次放大基本上避免了像素替换与零阶保持的缺点，可以实现任意比例的放大，其中 K 表示放大因子 factor。其工作步骤大致可以分为如下几步：

1) 取两个相邻像素，两个像素值相减取其绝对值作为输出 OP。

2) 对输出结果 OP 除以放大因子 (K) 得到步长 S，然后将 S 加到两个像素中像素值较小的那个，得到一个新像素值 (采样)，将新像素值插到上面提到的两个相邻像素之间。继续对新得到相邻像素像素值加上步长 S，生成新的像素，直到新像素的个数达到 K-1 个为止。

3) 重复上述步骤，在行与列方向即可得到放大以后的图像采样。

假设图像像素数组为 2×3 ，即两行三列，像素数据数组为：

15	30	15
30	15	30

行采样当 $K = 3$ 时，根据上面的第二步规则得到需要插入的新像素个数为 $2(K-1)$ ，说明要在 15 与 30 之间插入两个采样。对第一行 15 与 30 相减，结果为 15，除以 K 以后为 5，加上 15 与 30 中较小的像素值为 15，结果为 $15 + 5 = 20$ 。则第一个采样为 20，第二采样为 $20 + 5 = 25$ 。同样对剩下的像素进行相似处理，最终行采样放大以后的结果如下：

15	20	25	30	20	25	15
30	20	25	15	20	25	30

从上面可以看到当相邻两个像素值第一个比第二个大时，应该将采样得到的结果按照顺序重新调整为如下最终行采样结果：

15	20	25	30	25	20	15
30	25	20	15	20	25	30

同样对上面的行采样以后的结果进行列采样放大，得到的最终像素数组如下：

15	20	25	30	25	20	15
20	21	21	25	21	21	20
25	22	22	20	22	22	25
30	25	20	15	20	25	30

基于 K 次放大方法实现行采样的代码如下：

```
int k = (int) times;
int dh = k * (height-1) + 1;
int dw = k * (width-1) + 1;
```

```

int[] outPixels = new int[dw * dh];
int index = 0;
int[] rowPixels = new int[dw * height];
for (int row = 0; row < height; row++) {
    int ta = 0, tr = 0, tg = 0, tb = 0;
    for (int col = 1; col < width; col++) {
        index = row * width + col;
        ta = (inPixels[index] >> 24) & 0xff;
        tr = (inPixels[index] >> 16) & 0xff;
        tg = (inPixels[index] >> 8) & 0xff;
        tb = inPixels[index] & 0xff;
        int pcol = col - 1;
        if (pcol < 0)
        {
            pcol = 0;
        }
        int index2 = row * width + pcol;
        int ta2 = (inPixels[index2] >> 24) & 0xff;
        int tr2 = (inPixels[index2] >> 16) & 0xff;
        int tg2 = (inPixels[index2] >> 8) & 0xff;
        int tb2 = inPixels[index2] & 0xff;
        int optr = Math.abs(tr - tr2) / k;
        int optg = Math.abs(tg - tg2) / k;
        int optb = Math.abs(tb - tb2) / k;
        for (int t = 1; t < k; t++)
        {
            int ncol = col * k - (k - t);
            if (ncol < 0)
            {
                ncol = 0;
            }
            index = row * dw + ncol;
            int tr3 = Math.min(tr, tr2) + t * optr;
            int tg3 = Math.min(tg, tg2) + t * optg;
            int tb3 = Math.min(tb, tb2) + t * optb;
            rowPixels[index] = (ta << 24)
                | (tr3 << 16) | (tg3 << 8) | tb3;
        }
        index = row * dw + col * k;
        rowPixels[index] = (ta << 24)
            | (tr << 16) | (tg << 8) | tb;
        index = row * dw + (col * k) - k;
        rowPixels[index] = (ta << 24)
            | (tr2 << 16) | (tg2 << 8) | tb2;
    }
}

```

继续上述代码，实现列采样的代码如下：

```

// for each column
for (int col = 0; col < dw; col++) {

```



```

3. int ta = 0, tr = 0, tg = 0, tb = 0;
   for (int row = 1; row < height; row++) {
       index = row * dw + col;
       ta = (rowPixels[index] >> 24) & 0xff;
       tr = (rowPixels[index] >> 16) & 0xff;
       tg = (rowPixels[index] >> 8) & 0xff;
       tb = rowPixels[index] & 0xff;

       int prow = (row - 1) < 0 ? 0 : (row - 1);
       int index2 = prow * dw + col;
       int ta2 = (rowPixels[index2] >> 24) & 0xff;
       int tr2 = (rowPixels[index2] >> 16) & 0xff;
       int tg2 = (rowPixels[index2] >> 8) & 0xff;
       int tb2 = rowPixels[index2] & 0xff;

       int optr = Math.abs(tr - tr2)/k;
       int optg = Math.abs(tg - tg2)/k;
       int optb = Math.abs(tb - tb2)/k;

       for(int t=1; t<k; t++)
       {
           int nrow = row*k - (k-t);
           if(nrow < 0)
           {
               nrow = 0;
           }
           index = nrow *dw + col;
           int tr3 = Math.min(tr, tr2) + t * optr;
           int tg3 = Math.min(tg, tg2) + t * optg;
           int tb3 = Math.min(tb, tb2) + t * optb;
           outPixels[index] = (ta << 24)
               | (tr3 << 16) | (tg3 << 8) | tb3;
       }
       index = (row * k) * dw + col;
       outPixels[index] = (ta << 24)
           | (tr << 16) | (tg << 8) | tb;
       index = (row * k - k) * dw + col;
       outPixels[index] = (ta << 24)
           | (tr2 << 16) | (tg2 << 8) | tb2;
   }
}

```

笔者已经把三种图像放大的方法封装为类 `ZoomFilter.java`，调用该类实现图像放大的测试，只需要在 `ImagePanel` 的 `process()` 方法中添加如下几行代码即可：

```

ZoomFilter filter = new ZoomFilter();
filter.setType(ZoomFilter.K_TIMES_ZOOM);
filter.setTimes(3);
destImage = filter.filter(sourceImage, null);

```

运行 `MainUI`，选择一张图片，单击【处理】按钮即可查看效果，可以看到图像放大以后有很明显的边缘锯齿与模糊，在本章的下面内容将着力解决这一问题。



注意 如果没有特别说明, 所有算法类的测试都放在 ImagePanel.java 的 process 方法中, 然后运行 MainUI.java, 选择一张图片之后, 单击【处理】按钮查看运行效果, 这些在以后的章节中将不再重复说明。

7.2 临近点插值算法

1. 插值原理

本节重点介绍一种简单易用的图像插值算法——临近点插值算法。插值原理就是寻找离目标像素最近的源像素值, 可以很形象地表示为图 7-1。

其中的 X 表示目标图像需要插值的像素点 $D(x, y)$, 根据临近点插值原则, 它距离 $P(1, 0)$ 像素最近, 所以就用 $P(1, 0)$ 的像素值作为 $D(x, y)$ 的值。根据图 7-1 可知, 当一幅二维数字图像从源图像 $N \times M$ 被放为 $(j \times N) * (k \times M)$ 目标图像时, 参照数学斜率计算公式必然有:

P (0.0)	P (0.1)
P (1.0)	P (1.1)

图 7-1 临近点插值示意图

$$(X_1 - X_{\min}) / (X_{\max} - X_{\min}) = (Y_1 - Y_{\min}) / (Y_{\max} - Y_{\min})$$

当 X_{\min} 和 Y_{\min} 均为从零开始的像素点时, 公式可以简化为: $X = Y_1 (X_{\max} / Y_{\max})$

对于任意一幅源图像来说, 假设放大后目标图像的宽为 D_w , 高为 D_h , 任意目标像素点 (D_x, D_y) 在源图像上的位置为:

$$S_x = D_x (S_h / D_h) // \text{row}$$

$$S_y = D_y (S_w / D_w) // \text{column}$$

其中, (S_x, S_y) 为对于的源图像上的像素点, S_w 和 S_h 分别为源图像的宽度和高度。最终有:

$$D_{\text{pixel}}(D_x, D_y) = S_{\text{pixel}}(S_x, S_y);$$

2. 实现步骤与代码解析

实现临近点插值首先要根据源图像与目标图像的宽度与高度计算图像放缩比例, 然后根据目标图像像素点位置, 寻找其在源图像中的最近邻像素, 将该像素赋值给目标像素点。对目标像素所有像素点完成上述操作即实现了图像临近点插值放缩。

计算放缩比例的代码如下:

```
float rowRatio = ((float)height) / ((float)destH);
float colRatio = ((float)width) / ((float)destW);
```

根据目标像素点位置计算源像素点位置的代码如下:

```
int srcRow = Math.round(((float)row)*rowRatio);
if(srcRow >= height) {
    srcRow = height - 1;
}
// find the column index of source pixel
```

```
int srcCol = Math.round(((float) col) * colRatio);
if (srcCol >= width) {
    srcCol = width - 1;
}
```

像素赋值的代码如下：

```
outPixels[index2] = inPixels[index];
```

其中 outPixels 表示输出像素数组，inPixels 表示源图像像素数组。完整的基于临近点插值实现图像放大的类代码如下：

```
package com.book.chapter.seven;

import java.awt.image.BufferedImage;
import java.awt.image.ColorModel;

import com.book.chapter.four.AbstractBufferedImageOp;

public class NearestZoomFilter extends AbstractBufferedImageOp {

    private int destH; // zoom height
    private int destW; // zoom width

    public NearestZoomFilter() {
        System.out.println("Nearest Pixel Interpolation");
    }

    public void setDestHeight(int destH) {
        this.destH = destH;
    }

    public void setDestWidth(int destW) {
        this.destW = destW;
    }

    @Override
    public BufferedImage filter(BufferedImage src,
                               BufferedImage dest) {
        int width = src.getWidth();
        int height = src.getHeight();

        if (dest == null)
            dest = createCompatibleDestImage(src, null);

        int[] inPixels = new int[width * height];
        int[] outPixels = new int[destH * destW];
        getRGB(src, 0, 0, width, height, inPixels);
        float rowRatio = ((float) height) / ((float) destH);
```

```

float colRatio = ((float)width)/((float)destW);
int index = 0;
for (int row = 0; row < destH; row++) {
    int srcRow = Math.round(
        ((float)row)*rowRatio);
    if(srcRow >=height) {
        srcRow = height - 1;
    }
    for (int col = 0; col < destW; col++) {
        int srcCol = Math.round(
            ((float)col)*colRatio);
        if(srcCol >= width) {
            srcCol = width - 1;
        }
        int index2 = row * destW + col;
        index = srcRow * width + srcCol;
        outPixels[index2] = inPixels[index];
    }
    setRGB(dest, 0, 0, destW, destH, outPixels);
    return dest;
}

```

```

public BufferedImage createCompatibleDestImage(
    BufferedImage src, ColorModel dstCM) {
    if ( dstCM == null )
        dstCM = src.getColorModel();
    return new BufferedImage(dstCM,
        dstCM.createCompatibleWritableRaster(destW, destH),
        dstCM.isAlphaPremultiplied(), null);
}

```

其中重载了 `createCompatibleDestImage()` 方法目的是创建一个放大以后的 `BufferedImage` 对象。调用 `NearestZoomFilter` 实现图像放大，同样只需要在 `ImagePanel` 的 `process` 方法添加几行简单代码以后，运行 `MainUI.java` 选择需要的放大的图片，单击【处理】按钮即可查看效果。需要添加的代码如下：

```

NearestZoomFilter filter = new NearestZoomFilter();
filter.setDestHeight(sourceImage.getHeight() * 2);
filter.setDestWidth(sourceImage.getWidth() * 2);
destImage = filter.filter(sourceImage, null);

```

临近点插值是一种快速放大的方法，不足之处是基于该方法产生图像有明显的锯齿和模糊，不是一种很好的插值算法。

7.3 双线性插值算法

1. 插值原理

本节将介绍一种反锯齿的图像插值算法——双线性插值，该方法基于对插值像素周围四个源像素点值与之距离权重作为考量依据从而计算得到插值像素点的像素值。可以将其形象地表示为图 7-2。

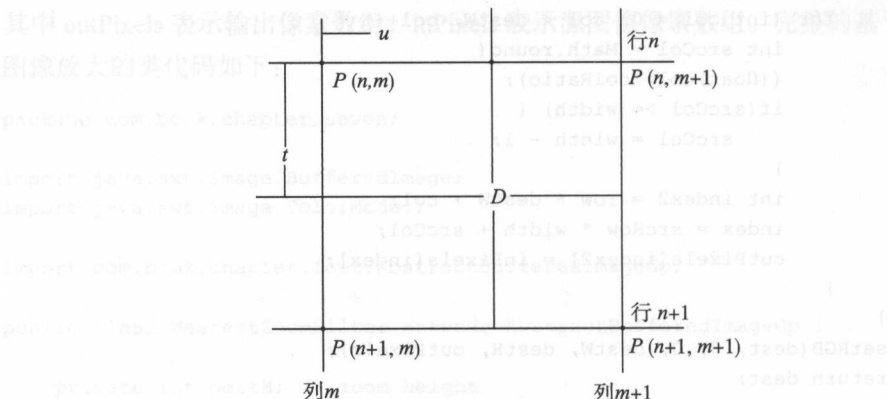


图 7-2 双线性插值示意图

其中 $D(t, u)$ 表示采样点的小数部分坐标，矩形四个角点分别表示四个像素点 $P(n, m)$ 、 $P(n+1, m)$ 、 $P(n, m+1)$ 、 $P(n+1, m+1)$ 。从图 7-2 中可以看出，采样点 D 水平方向距离 $P(n, m)$ 与 $P(n+1, m)$ 的距离为 u ，则距离 $P(n, m+1)$ 与 $P(n+1, m+1)$ 的距离可以表示为 $1-u$ ，通常情况下，距离越远则权重越小，距离越近意味着对采样像素贡献越大，权重也应该越大，所以当距离为 u 时，对应权重应为 $1-u$ 。由此可以得到两个水平方向权重像素：

$$T_1 = (1-u) \times P(n, m) + u \times P(n, m+1)$$

$$T_2 = (1-u) \times P(n+1, m) + u \times P(n+1, m+1)$$

根据 T_1 、 T_2 值与垂直方向的权重系数 u 和 $1-u$ ，可以得到最终采样像素值：

$$D(x, y) = (1-t) T_1 + t \times T_2$$

将 T_1 与 T_2 带入，最终得到如下：

$$\begin{aligned} D(x, y) = & P(n, m) \times (1-t) \times (1-u) + P(n, m+1) \times u \times (1-t) + P(n+1, m) \\ & \times (1-u) \times t + P(n+1, m+1) \times t \times u \end{aligned}$$

根据该公式即可计算出目标采样像素值。根据双线性插值的基本原理，程序实现双线性插值可以分为如下几步：

- 1) 获取源图像像素数组，根据放缩比率获得目标图像的宽与高。
- 2) 循环目标图像上的每个像素，根据坐标寻找其在源图像中的四个相邻像素。
- 3) 根据小数部分坐标计算得到的像素值即为目标图像像素值。

2. 编码实现

计算放缩比率的代码如下：

```
float rowRatio = ((float)height)/((float)destH);
float colRatio = ((float)width)/((float)destW);
```

根据比例计算目标像素 Y 方向整数与小数坐标的代码如下：

```
double srcRow = ((float)row)*rowRatio;
// 获取整数部分坐标 row Index
double j = Math.floor(srcRow);
// 获取行的小数部分坐标
double t = srcRow - j;
```

计算 X 方向整数与小数坐标的代码如下：

```
double srcCol = ((float)col)*colRatio;
// 获取整数部分坐标 column Index
double k = Math.floor(srcCol);
// 获取列的小数部分坐标
double u = srcCol - k;
```

根据坐标在源像素数组中获取四个相邻像素点的代码如下：

```
int[] p1 = getPixel(j, k, width, height, inPixels);
int[] p2 = getPixel(j, k+1, width, height, inPixels);
int[] p3 = getPixel(j+1, k, width, height, inPixels);
int[] p4 = getPixel(j+1, k+1, width, height, inPixels);
```

根据公式计算得到目标像素值的代码如下：

```
double a = (1.0d-t)*(1.0d-u);
double b = (1.0d-t)*u;
double c = (t)*(1.0d-u);
double d = t*u;
ta = 255;
tr = (int)(p1[0] * a + p2[0] * b + p3[0] * c + p4[0] * d);
tg = (int)(p1[1] * a + p2[1] * b + p3[1] * c + p4[1] * d);
tb = (int)(p1[2] * a + p2[2] * b + p3[2] * c + p4[2] * d);
```

实现双线性插值完整的代码如下：

```
package com.book.chapter.seven;

import java.awt.image.BufferedImage;
import java.awt.image.ColorModel;

import com.book.chapter.four.AbstractBufferedImageOp;

public class BilinearZoomFilter extends AbstractBufferedImageOp {
    private int destH; // zoom height
    private int destW; // zoom width
    public BilinearZoomFilter()
```


7.3 双线性插值算法

```

    }

    public void setDestHeight(int destH) {
        this.destH = destH;
    }

    public void setDestWidth(int destW) {
        this.destW = destW;
    }

    @Override
    public BufferedImage filter(BufferedImage src, BufferedImage dest) {
        int width = src.getWidth();
        int height = src.getHeight();

        if (dest == null)
            dest = createCompatibleDestImage(src, null);

        int[] inPixels = new int[width * height];
        int[] outPixels = new int[destH * destW];
        getRGB(src, 0, 0, width, height, inPixels);
        float rowRatio = ((float)height)/((float)destH);
        float colRatio = ((float)width)/((float)destW);
        int index = 0;
        for(int row=0; row<destH; row++) {
            int ta = 0, tr = 0, tg = 0, tb = 0;
            double srcRow = ((float)row)*rowRatio;
            // 获取整数部分坐标 row Index
            double j = Math.floor(srcRow);
            // 获取行的小数部分坐标
            double t = srcRow - j;
            for(int col=0; col<destW; col++) {
                double srcCol = ((float)col)*colRatio;
                // 获取整数部分坐标 column Index
                double k = Math.floor(srcCol);
                // 获取列的小数部分坐标
                double u = srcCol - k;
                int[] p1 = getPixel(j, k, width, height, inPixels);
                int[] p2 = getPixel(j, k+1, width, height, inPixels);
                int[] p3 = getPixel(j+1, k, width, height, inPixels);
                int[] p4 = getPixel(j+1, k+1, width, height, inPixels);
                double a = (1.0d-t)*(1.0d-u);
                double b = (1.0d-t)*u;
                double c = (t)*(1.0d-u);
                double d = t*u;
                ta = 255;
                tr = (int)(p1[0] * a + p2[0] * b + p3[0] * c + p4[0] * d);
                tg = (int)(p1[1] * a + p2[1] * b + p3[1] * c + p4[1] * d);
                tb = (int)(p1[2] * a + p2[2] * b + p3[2] * c + p4[2] * d);
                index = row * destW + col;
                outPixels[index] = (ta << 24)
            }
        }
    }

```

```

        | (clamp(tr) << 16) | (clamp(tg) << 8) | clamp(tb);
    }
}

setRGB(dest, 0, 0, destW, destH, outPixels);
return dest;
}

private int[] getPixel(double j, double k,
    int width,
    int height,
    int[] inPixels) {
    int row = (int)j;
    int col = (int)k;
    if(row >= height)
    {
        row = height - 1;
    }
    if(row < 0)
    {
        row = 0;
    }
    if(col < 0)
    {
        col = 0;
    }
    if(col >= width)
    {
        col = width - 1;
    }
    int index = row * width + col;
    int[] rgb = new int[3];
    rgb[0] = (inPixels[index] >> 16) & 0xff;
    rgb[1] = (inPixels[index] >> 8) & 0xff;
    rgb[2] = inPixels[index] & 0xff;
    return rgb;
}

public BufferedImage createCompatibleDestImage(
    BufferedImage src, ColorModel dstCM) {
    if (dstCM == null)
        dstCM = src.getColorModel();
    return new BufferedImage(dstCM,
        dstCM.createCompatibleWritableRaster(
            destW, destH),
        dstCM.isAlphaPremultiplied(), null);
}
}

```

运行与测试 `BilinearZoomFilter`，只需要在 `ImagePanel` 的 `process` 方法中添加如下几行代码：

```

BilinearZoomFilter filter = new BilinearZoomFilter();
filter.setDestHeight(sourceImage.getHeight() * 2);
filter.setDestWidth(sourceImage.getWidth() * 2);
destImage = filter.filter(sourceImage, null);

```

然后在 eclipse 中运行 MainUI.java，选择一张图像单击【处理】按钮即可查看效果。

3. 双线性插值优缺点

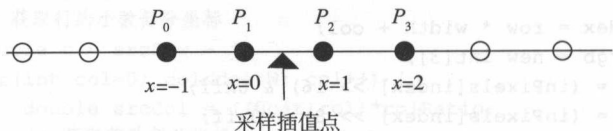
双线性内插值算法在图像的放缩处理中具有抗锯齿功能，是最简单和常见的图像放缩算法，但是双线性内插值算法没有考虑边缘和图像的梯度变化，相比之下，双立方插值算法可以更好地解决这些问题。下面一节介绍双立方插值的相关知识。

7.4 双立方插值与 Lanczos 采样

本节介绍两种经典的图像插值算法——双立方插值算法与 Lanczos 采样插值算法，相比之前介绍的插值算法，这两种插值算法计算量更大，插值效果也更好，在许多图像处理软件中被广泛应用，因此更具工程实践性与重要性。

7.4.1 双立方插值算法

双立方插值是一种高精度的插值算法，当然其计算复杂性也远远超过双线性插值。双立方插值考虑采样像素周围 16 个源像素值，然后根据权重系数进行计算，最终得到采样像素点像素值。在具体介绍双立方插值的原理之前，首先看一下一维的双立方插值图示：



三角箭头表示采样插值点 D ， P_0 、 P_1 、 P_2 、 P_3 表示四个像素点。假设三次多项式为 $f(x) = ax^3 + bx^2 + cx + d$ ，其导数为

$$f(x) = 3ax^2 + 2bx + c$$

其在 $x = 0$ 与 $x = 1$ 处的值与其导数的值分别为：

$$f(0) = d$$

$$f(1) = a + b + c + d$$

$$f'(0) = c$$

$$f'(1) = 3a + 2b + c$$

根据上述四个方程式，可以得到四个参数值分别为：

$$a = 2f(0) - 2f(1) + f'(0) + f'(1)$$

$$b = -3f(0) + 3f(1) - 2f'(0) - f'(1)$$

$$c = f'(0)$$

$$d = f(0)$$

根据上述插值公式可知, 点 $x = 0$ 时值为 p_1 , $x = 1$ 时值为 p_2 , 所以 $f(0) = p_1$ 、 $f(1) = p_2$ 。根据导数在离散点上的特征 (就是它们之间的差值), 可得到:

$$f'(0) = \frac{p_2 - p_0}{2}, f'(1) = \frac{p_3 - p_1}{2}$$

代入 $f(0)$ 、 $f(1)$ 及其对应导数值即可获得 a 、 b 、 c 、 d 四个参数的值, 从而得到采样点 D 的值, 可以表达为:

$$f(p_0, p_1, p_2, p_3, x) = \left(-\frac{1}{2}p_0 + \frac{3}{2}p_1 - \frac{3}{2}p_2 + \frac{1}{2}p_3\right)x^3 + \left(p_0 - \frac{5}{2}p_1 + 2p_2 - \frac{1}{2}p_3\right)x^2 + \left(-\frac{1}{2}p_0 + \frac{1}{2}p_2\right)x + p_1$$

这里要特别注意, x 是 D 的小数坐标。对于二维的图像数组, 双立方插值有 $4 \times 4 = 16$ 个像素点, 首先计算每一行的 4 个像素点, 最后计算列的 4 个, 数学公式表示如下:

$$G(x, y) = f(f(p_{00}, p_{01}, p_{02}, p_{03}, y), f(p_{10}, p_{11}, p_{12}, p_{13}, y), f(p_{20}, p_{21}, p_{22}, p_{23}, y), f(p_{30}, p_{31}, p_{32}, p_{33}, y), x)$$

其中 x 、 y 表示采样点 G 的小数部分坐标。二维的双立方插值可以表示为图 7-3。

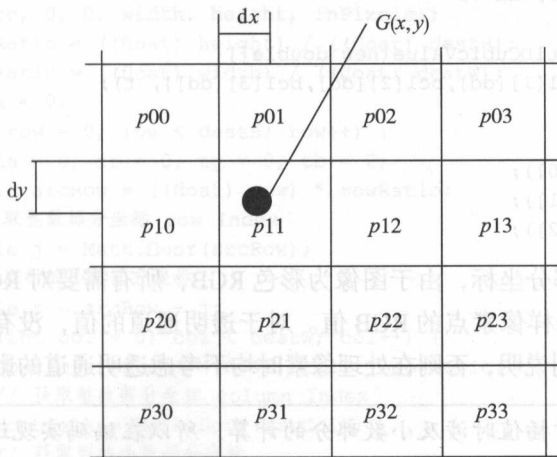


图 7-3 双立方插值示意图

上面就是双立方插值的基本原理, 双立方插值基本的数学原理基于三次多项式插值公式。根据上述知识, 实现双立方插值代码就会变得很容易。下面剖析其中关键的编程代码。

一维双立方插值计算代码片段如下 (请对照公式阅读):

```
// extract (1/2*delta) will get below formula
public double get1DCubicValue(double[] p, double delta) {
```

```

return p[1]
    + 0.5
    * delta
    * (p[2] - p[0]
    + delta * (2.0 * p[0] - 5.0 * p[1] + 4.0 * p[2] - p[3]
    + delta * (3.0 * (p[1] - p[2]) + p[3] - p[0])));
}

```

利用一维双立方插值计算结果，得到二维双立方插值计算结果的代码如下：

```

for(int n=0; n<4; n++)
{
    int[][] c1 = new int[4][3];
    for(int m=0; m<4; m++)
    {
        c1[m] = getPixel(j+n-1, k+m-1,
            width, height, inPixels);
    }
    for(int d=0; d<3; d++) // for RGB
    {
        bc1[n][d] = get1DCubicValue(new double[]{
            c1[0][d], c1[1][d], c1[2][d], c1[3][d]}, u);
    }
}
double[] dRGB = new double[3];
for(int dd=0; dd<3; dd++)
{
    dRGB[dd] = get1DCubicValue(new double[]{
        bc1[0][dd], bc1[1][dd], bc1[2][dd], bc1[3][dd]}, t);
}
ta = 255;
tr = (int) (dRGB[0]);
tg = (int) (dRGB[1]);
tb = (int) (dRGB[2]);

```

其中 t , u 为小数部分坐标，由于图像为彩色 RGB，所有需要对 RGB 三个分量都做相似处理才可以得到最终采样像素点的 RGB 值。对于透明通道的值，没有做处理，一直默认为 255，本书中除非作特别说明，否则在处理像素时均不考虑透明通道的影响。



注意 双线性与双立方插值时涉及小数部分的计算，所以在编码实现过程中要特别注意保留计算结果小数部分精度，这样才会使计算结果更加准确。

最终实现双立方插值算法的类 BicubicZoomFilter 的完整源代码如下：

```

package com.book.chapter.seven;

import java.awt.image.BufferedImage;
import java.awt.image.ColorModel;

import com.book.chapter.four.AbstractBufferedImageOp;

```

```

public class BicubicZoomFilter extends AbstractBufferedImageOp {
    private int destH; // zoom height
    private int destW; // zoom width
    public BicubicZoomFilter()
    {
    }
    public void setDestHeight(int destH) {
        this.destH = destH;
    }
    public void setDestWidth(int destW) {
        this.destW = destW;
    }
    @Override
    public BufferedImage filter(BufferedImage src, BufferedImage dest) {
        int width = src.getWidth();
        int height = src.getHeight();

        if (dest == null)
            dest = createCompatibleDestImage(src, null);

        int[] inPixels = new int[width * height];
        int[] outPixels = new int[destH * destW];
        getRGB(src, 0, 0, width, height, inPixels);
        float rowRatio = ((float) height) / ((float) destH);
        float colRatio = ((float) width) / ((float) destW);
        int index = 0;
        for (int row = 0; row < destH; row++) {
            int ta = 0, tr = 0, tg = 0, tb = 0;
            double srcRow = ((float) row) * rowRatio;
            // 获取整数部分坐标 row Index
            double j = Math.floor(srcRow);
            // 获取行的小数部分坐标
            double t = srcRow - j;
            for (int col = 0; col < destW; col++) {
                double srcCol = ((float) col) * colRatio;
                // 获取整数部分坐标 column Index
                double k = Math.floor(srcCol);
                // 获取列的小数部分坐标
                double u = srcCol - k;
                double[][] bcl = new double[4][3];
                for (int n = 0; n < 4; n++) {
                    int[][] c1 = new int[4][3];
                    for (int m = 0; m < 4; m++) {
                        c1[m] = getPixel(j+n-1, k+m-1, width, height, inPixels);
                    }
                    for (int d = 0; d < 3; d++) // for RGB

```



```

        {
            bcl[n][d] = get1DCubicValue(new double[]
                {c1[0][d], c1[1][d], c1[2][d], c1[3][d]}, u);
        }
    }
    double[] dRGB = new double[3];
    for(int dd=0; dd<3; dd++)
    {
        dRGB[dd] = get1DCubicValue(new double[]
            {bcl[0][dd], bcl[1][dd], bcl[2][dd], bcl[3][dd]}, t);
    }
    ta = 255;
    tr = (int) (dRGB[0]);
    tg = (int) (dRGB[1]);
    tb = (int) (dRGB[2]);
    index = row * destW + col;
    outPixels[index] = (ta << 24) | (clamp(tr)<< 16)
        | (clamp(tg) << 8) | clamp(tb);
    }
}
setRGB(dest, 0, 0, destW, destH, outPixels);
return dest;
}

// extract (1/2*delta) will get below formula
public double get1DCubicValue(double[] p, double delta){
    return p[1]
        + 0.5
        * delta
        * (p[2] - p[0]
            + delta * (2.0 * p[0] - 5.0 * p[1] + 4.0 * p[2] - p[3]
                + delta * (3.0 * (p[1] - p[2])) + p[3] - p[0])));
}

private int[] getPixel(double j, double k, int width, int height,
    int[] inPixels) {
    int row = (int) j;
    int col = (int) k;
    if (row >= height) {
        row = height - 1;
    }
    if (row < 0) {
        row = 0;
    }
    if (col < 0) {
        col = 0;
    }
    if (col >= width) {
        col = width - 1;
    }
    int index = row * width + col;

```

```

int[] rgb = new int[3];
rgb[0] = (inPixels[index] >> 16) & 0xff;
rgb[1] = (inPixels[index] >> 8) & 0xff;
rgb[2] = inPixels[index] & 0xff;
return rgb;
}

public BufferedImage createCompatibleDestImage(
    BufferedImage src, ColorModel dstCM) {
    if (dstCM == null)
        dstCM = src.getColorModel();
    return new BufferedImage(dstCM,
        dstCM.createCompatibleWritableRaster(destW, destH),
        dstCM.isAlphaPremultiplied(), null);
}
}

```

同样，测试运行该类只需要如下几行代码即可：

```

BicubicZoomFilter filter = new BicubicZoomFilter();
filter.setDestHeight(sourceImage.getHeight() * 2);
filter.setDestWidth(sourceImage.getWidth() * 2);
destImage = filter.filter(sourceImage, null);

```

如果仔细观察放大以后的图像就会发现，相比源图像，放大以后的图像好像有点模糊，没错，事实就是如此，无论是基于三次方多项式还是 B 样条曲线实现的双立方插值，都叫做双立方 blur 版本 (bi-cubic blur)，与之相对应的是基于 CatMull-Rom 采样实现的双立方插值叫做双立方 sharpen 版本 (bi-cubic sharpen)。下面就来介绍如何实现双立方插值的 sharpen 版本。首先看一下 CatMull-Rom 的数学表达式：

$$K(x) = \frac{1}{6} \begin{cases} (12 - 9B - 6C) |x|^3 + (-18 + 12B + 6C) |x|^2 + (6 - 2B), & \text{如果 } |x| < 1 \\ (-B - 6C) |x|^3 + (6B + 30C) |x|^2 + (12B - 48C) |x| + (8B + 24C), & \text{如果 } 1 \leq |x| < 2 \\ 0, & \text{其他情况下} \end{cases}$$

同时，针对双立方插值，有更加简化与形象的理解，就是计算 16 个像素的权重系数与其对应的像素值的乘积，所以双立方插值还可以简化为如下数学公式：

$$P'(x, y) = \sum_{m=-1}^2 \sum_{n=-1}^2 P(x+m, y+n) R_c\{(m-a)\} R_c\{-(n-b)\}$$

其中 a 、 b 分别为采样插值点的小数部分坐标， $P(x, y)$ 是源像素值。 R_c 表示插值采样权重计算公式，这里是指 CatMull-Rom 方法。根据上述这些知识，基于 CatMull-Rom 实现双立方插值 sharpen 版的关键在于如何计算 CatMull-Rom 的权重，根据公式实现权重计算的代码如下：

```

private double CatMullRom( double f )
{
    if( f < 0.0 )
    {
        f = Math.abs(f);
    }
    if( f < 1.0 )
    {
        return ((12-9*B-6*C)*(f*f*f)+(-18+12*B+6*C)*(f*f)+(6-2*B))/6.0;
    }
    else if( f >= 1.0 && f < 2.0 )
    {
        return ((-B-6*C)*(f*f*f)+(6*B+30*C)*(f*f)+(- (12*B)-48*C)*f+8*B+24*C)/ 6.0;
    }
    else
    {
        return 0.0;
    }
}

```

根据采样点小数坐标计算最终采样像素的代码片段如下：

```

double[] rgbData = new double[3];
double rgbCoffeData = 0.0;
for(int m=-1; m<3; m++)
{
    for(int n=-1; n<3; n++)
    {
        int[] rgb = getPixel(j+m, k+n,
                               width, height, inPixels);
        double f1 = CatMullRom( ((double) m ) - t );
        double f2 = CatMullRom( -(( (double) n ) - u ) );
        // sum of weight
        rgbCoffeData += f2*f1;
        // sum of the RGB values
        rgbData[0] += rgb[0] * f2 * f1;
        rgbData[1] += rgb[1] * f2 * f1;
        rgbData[2] += rgb[2] * f2 * f1;
    }
}
ta = 255;
// get Red/green/blue value for sample pixel
tr = (int) (rgbData[0]/rgbCoffeData);
tg = (int) (rgbData[1]/rgbCoffeData);
tb = (int) (rgbData[2]/rgbCoffeData);
index = row * destW + col;
outPixels[index] = (ta << 24) | (clamp(tr) << 16)
                    | (clamp(tg) << 8) | clamp(tb);

```



注意 在上述代码中，计算权重和是为了实现像素归一化，使像素值始终在 0~255 之间，这个是很多图像处理算法中非常重要的编程技巧，但是又从来不体现在相关数学知识介

绍中，完全是实践中的经验总结。

完整的基于 CatMull-Rom 公式实现双立方插值的类 `BicubicSharpenFilter.java` 的代码请下载阅览，源代码已经经过严格测试，请读者一定认真阅读并实践，同时源代码也是本书的一部分，可以帮助读者加深对理论知识的认知、理解与领悟。关于双立方插值的介绍就到这里，对此感兴趣的读者可以进一步研究。

7.4.2 Lanczos 采样插值算法

Lanczos 采样算法主要用来在数字信号之间进行插值采样，它是基于 Lanczos 核与窗口两个参数来计算采样值的，然后还会基于 \sin 三角函数值计算权重，带入得到最终的像素值，窗口大小表示采样插值需要考虑的像素多少。具体的数学表达式如下：

$$L(x) = \begin{cases} 1 & x=0 \\ \frac{a \sin(\pi x) \sin(\pi x/a)}{\pi^2 x^2} & 0 < |x| < a \\ 0 & |x| \geq a \end{cases}$$

其中 $x=0$ 时， L 值为 1，当 $x>a$ 时， L 值为 0，当 $0 < |x| < a$ 时，计算权重。

值得注意的是，这里的 a 指的是 Lanczos 的窗口大小。基于 Lanczos 采样插值算法的大致流程如下：

- 1) 计算图像放缩比例 `ratio`。
- 2) 计算采样像素，计算整数与浮点数坐标。
- 3) 计算 Lanczos 窗口中每个源像素点权重。
- 4) 根据权重计算加权平均值得到新像素值。
- 5) 对每个采样像素重复 2 ~ 4 步，即得到结果像素数组。

根据上述知识，实现 Lanczos 采样放缩算法的代码如下：

```
package com.book.chapter.seven;

import java.awt.image.BufferedImage;
import java.awt.image.ColorModel;

import com.book.chapter.four.AbstractBufferedImageOp;

public class LanczosZoomFilter extends AbstractBufferedImageOp {
    // lanczos_size
    private float lanczosSize;
    private float destWidth;

    public LanczosZoomFilter() {
        lanczosSize = 3;
        destWidth = 100;
    }
}
```

```

    public LanczosZoomFilter(float lobes, int width) {
        this.lanczosSize = lobes;
        this.destWidth = width;
    }

    public void setLanczosSize(float size) {
        this.lanczosSize = size;
    }

    public void setDestWidth(float destWidth) {
        this.destWidth = destWidth;
    }

    @Override
    public BufferedImage filter(BufferedImage src, BufferedImage dest) {
        int width = src.getWidth();
        int height = src.getHeight();
        float ratio = width / this.destWidth;
        float rcp_ratio = 2.0f / ratio;
        float range2 = (float) Math.ceil(ratio * lanczosSize / 2);

        // destination image
        int dh = (int) (height * (this.destWidth / width));
        int dw = (int) this.destWidth;

        if (dest == null) {
            ColorModel cMD = src.getColorModel();
            dest = new BufferedImage(src.getColorModel(),
                                     cMD.createCompatibleWritableRaster(dw, dh),
                                     cMD.isAlphaPremultiplied(), null);
        }

        int[] inPixels = new int[width * height];
        int[] outPixels = new int[dw * dh];

        getRGB(src, 0, 0, width, height, inPixels);
        int index = 0;
        float fcy = 0, icy = 0, fcx = 0, icx = 0;
        for (int row = 0; row < dh; row++) {
            int ta = 0, tr = 0, tg = 0, tb = 0;
            fcy = (row + 0.5f) * ratio;
            icy = (float) Math.floor(fcy);
            for (int col = 0; col < dw; col++) {
                fcx = (col + 0.5f) * ratio;
                icx = (float) Math.floor(fcx);

                float sumred = 0, sumgreen = 0, sumblue = 0;
                float totalWeight = 0;
                for (int subcol = (int) (icx - range2);

```

```

        subcol <= icx + range2; subcol++) {
            if (subcol < 0 || subcol >= width)
                continue;
            int ncol = (int) Math.floor(1000 *
                Math.abs(subcol - fcx));

            for (int subrow = (int) (icy - range2);
                subrow <= icy + range2; subrow++) {
                if (subrow < 0 || subrow >= height)
                    continue;
                int nrow = (int) Math.floor(1000 *
                    Math.abs(subrow - fcy));
                float weight = (float) getLanczosFactor(
                    Math.sqrt(Math
                        .pow(ncol * rcp_ratio, 2)
                        + Math.pow(nrow * rcp_ratio, 2)) / 1000);
                if (weight > 0) {
                    index = (subrow * width + subcol);
                    tr = (inPixels[index] >> 16) & 0xff;
                    tg = (inPixels[index] >> 8) & 0xff;
                    tb = inPixels[index] & 0xff;
                    totalWeight += weight;
                    sumred += weight * tr;
                    sumgreen += weight * tg;
                    sumblue += weight * tb;
                }
            }
            index = row * dw + col;
            tr = (int) (sumred / totalWeight);
            tg = (int) (sumgreen / totalWeight);
            tb = (int) (sumblue / totalWeight);
            outPixels[index] = (255 << 24) | (clamp(tr) << 16)
                | (clamp(tg) << 8) | clamp(tb);
            // clear for next pixel
            sumred = 0;
            sumgreen = 0;
            sumblue = 0;
            totalWeight = 0;
        }
    }
    setRGB(dest, 0, 0, dw, dh, outPixels);
    return dest;
}

```

```

private double getLanczosFactor(double distance) {
    if (distance > lanczosSize)
        return 0;
    distance *= Math.PI;

```



```

        if (Math.abs(distance) < 1e-16)
            return 1;
        double xx = distance / lanczosSize;
        return Math.sin(distance) * Math.sin(xx) / distance / xx;
    }
}

```

运行与测试 LanczosZoomFilter.java 时，只需要在 ImagePanel 类的 process 方法中添加如下代码片段即可：

```

LanczosZoomFilter filter = new LanczosZoomFilter();
destImage = filter.filter(sourceImage, null);

```

LanczosZoomFilter 中还涉及了如何计算权重 x 的问题，取像素点坐标所在行与列的算术平方根之后的值，除以常量 1000 得到权重 x ，其中 1000 是计算权重 x 时使用的经验值，当窗口大小不变，感兴趣的读者可以通过尝试不同的常量取值看看得到的结果有什么不同，更多的发现等待着读者动手去实践。

本节通过介绍两种不同的图像高精度的插值算法与代码实现，帮助读者厘清插值算法的理论与实际应用之间的联系和区别，从而对图像插值有更加深刻的认知与理解。当然每一种插值算法都不是完美的，它们有各自的缺点与优点，正所谓寸有所长、尺有所短。

7.5 图像旋转

图像旋转也是实际项目中经常遇到的问题，几乎所有主流的编程语言都支持通过 API 实现图像错切与旋转，本节将从图像旋转的基本原理、涉及问题、快速方法等几个方面来介绍图像旋转算法的编程实践。通过本节知识学习，读者可以轻松实现图像的错切与旋转，不再对此有任何认知上的盲点与难点。本节首先介绍一种通过中心极坐标变换双线性插值实现图像旋转的算法，然后介绍通过错切实现图像旋转的算法。此外，还会涉及一些简单而且基础的数学知识。

1. 基于线性插值的旋转方法

基于线性插值的图像旋转方法是一种采样插值图像旋转方法，当把图像的像素二维数组看成一个矩阵时，这里假设旋转都是围绕图像的中心点进行的，轴旋转角度为 θ ，这样旋转点与旋转角度就是已知值了，通过反三角函数计算像素各点距离中心点的距离 r ，然后根据反三角函数与极坐标相关知识就可以通过旋转以后的图像像素坐标计算得到源像素中四个最近像素点，从而实现双线性插值，得到旋转以后的图像像素点，最终完成图像旋转。二维的矩阵旋转公式如下：

$$\begin{bmatrix} \cos \alpha & -\sin \alpha \\ \sin \alpha & \cos \alpha \end{bmatrix}$$

假设源像素点 P 坐标为 (x_1, y_1) , 旋转中心点 C 坐标为 (x_0, y_0) , 则旋转角度 θ 以后得到的 P 点坐标为:

$$x_2 = \cos(\theta) \times (x_1 - x_0) - \sin(\theta) \times (y_1 - y_0) + x_0$$

$$y_2 = \sin(\theta) \times (x_1 - x_0) + \cos(\theta) \times (y_1 - y_0) + y_0$$

坐标 P 旋转以后的新位置为 (x_2, y_2) 。根据上述公式完成旋转以后的图片与原图相比不是很清晰, 其主要原因在于没有考虑图像新位置的小数部分对图像像素的影响, 而基于双线性插值旋转正好解决了旋转中的这个问题, 从而得到质量较高的旋转图片。前面计算双线性插值时, 我们通过目标像素反向寻找了源像素, 这里也是一样, 假设旋转以后的像素点 $P(x, y)$ 旋转角度为 θ , 根据像素点 P 到中心点的欧几里得距离 r 与角度 θ , 最终得到源像素点小数与整数坐标, 然后使用双线性插值算法得到像素值赋给旋转以后的像素点 P 。对旋转以后的每个像素点都做这样的处理, 就可完成图像的双线性插值旋转, 基本原理介绍到这里结束。

下面来剖析实现双线性插值旋转编码过程中的四个细节问题, 首先是特殊角度旋转编码的实现, 其次是如何通过旋转后的坐标寻找源像素中的对应像素点坐标, 然后是双线性插值得到像素值的方法, 最后是旋转以后非图像内容像素(背景像素)颜色填充。

(1) 特殊角度旋转

当角度为 90° 180° 、 270° 时, 代码如下:

```
int index = 0;
int index2 = 0;
if(specialAngle == 90)
{
    outw = height;
    outh = width;
}
else if(specialAngle == 180)
{
    outw = width;
    outh = height;
}
else if(specialAngle == 270)
{
    outw = height;
    outh = width;
}

int[] outPixels = new int[width * height];
for(int row=0; row<height; row++) {
    int ta = 0, tr = 0, tg = 0, tb = 0;
    for(int col=0; col<width; col++) {
        index = row * width + col;
        ta = (inPixels[index] >> 24) & 0xff;
        tr = (inPixels[index] >> 16) & 0xff;
        tg = (inPixels[index] >> 8) & 0xff;
        tb = inPixels[index] & 0xff;
```

```

if(specialAngle == 90)
{
    index2 = outw * col + (height - 1 - row);
}
else if(specialAngle == 180)
{
    index2 = outw * (height - 1 - row) +
        (width - 1 - col);
}
else if(specialAngle == 270)
{
    index2 = outw * (width - 1 - col) + row;
}
outPixels[index2] = (ta << 24) |
    (tr << 16) | (tg << 8) | tb;
}
}

BufferedImage dest = createCompatibleDestImage( src, null );
setRGB( dest, 0, 0, outw, outh, outPixels );
return dest;

```

(2) 源像素坐标计算

当旋转角度不是特殊角度时就需要计算源像素坐标，通过反三角函数知识与极坐标知识实现源像素坐标计算。首先要根据源图像宽与高计算目标图像宽与高，代码如下：

```

outw = (int)(width*Math.cos(angle)+height*Math.sin(angle));
outh = (int)(height*Math.cos(angle)+width*Math.sin(angle));

```

其次需要计算源图像与旋转后图像旋转中心位置，代码片段如下：

```

// calculate new center coordinate
float centerX = outw / 2.0f + 0.5f;
float centerY = outh / 2.0f + 0.5f;

// calculate the original center coordinate
float ocenterX = width / 2.0f + 0.5f;
float ocenterY = height / 2.0f + 0.5f;

```

然后计算旋转后图像像素点到中心点的距离，且根据距离计算与源像素点之间的角度，代码片段如下：

```

rx = col - centerX;
ry = centerY - row;
float fDistance = (float)Math.sqrt(rx * rx + ry * ry);
float fPolarAngle = 0; //;
if(rx != 0) {
    fPolarAngle = (float)Math.atan2((double)ry, (double)rx);
} else {
    if(rx == 0) {

```

```

    if(ry == 0) {
        outPixels[index] = centerPixel;
        continue;
    }
    else if(ry < 0) {
        fPolarAngle = 1.5f * (float)Math.PI;
    } else {
        fPolarAngle = 0.5f * (float)Math.PI;
    }
}
}

```

其中 $rx=0$ 或 $ry=0$ 表示特殊角度, 已经处理。最后根据角度集合三角函数知识得到像素点坐标, 代码片段如下:

```

// "reverse" rotate, so minus instead of plus
fPolarAngle -= angle;
px = fDistance * (float)Math.cos(fPolarAngle);
py = fDistance * (float)Math.sin(fPolarAngle);

// get original pixel float point
prow = ((float)ocenterY) - py;
pcol = ((float)ocenterX) + px;

```

(3) 双线性插值

根据上面获得的浮点数坐标, 实现双线性插值, 代码片段如下:

```

private int[] bilineInterpolation(int[] input,
    int width, int height,
    float prow, float pcol) {
    double row = Math.floor(prow);
    double col = Math.floor(pcol);
    if(row < 0 || row >= height) {
        return new int[]{background.getRed(),
            background.getGreen(), background.getBlue()};
    }
    if(col < 0 || col >= width) {
        return new int[]{background.getRed(),
            background.getGreen(), background.getBlue()};
    }

    int rowNext = (int)row + 1, colNext = (int)col + 1;
    if((row + 1) >= height) {
        rowNext = (int)row;
    }
    if((col + 1) >= width) {
        colNext = (int)col;
    }

    double t = prow - row;
    double u = pcol - col;

```

```

double coffiecent1 = (1.0d-t)*(1.0d-u);
double coffiecent2 = (t)*(1.0d-u);
double coffiecent3 = t*u;
double coffiecent4 = (1.0d-t)*u;

int index1 = (int)(row * width + col);
int index2 = (int)(row * width + colNext);

int index3 = (int)(rowNext * width + col);
int index4 = (int)(rowNext * width + colNext);
int tr1, tr2, tr3, tr4;
int tg1, tg2, tg3, tg4;
int tb1, tb2, tb3, tb4;

tr1 = (input[index1] >> 16) & 0xff;
tg1 = (input[index1] >> 8) & 0xff;
tb1 = input[index1] & 0xff;

tr2 = (input[index2] >> 16) & 0xff;
tg2 = (input[index2] >> 8) & 0xff;
tb2 = input[index2] & 0xff;

tr3 = (input[index3] >> 16) & 0xff;
tg3 = (input[index3] >> 8) & 0xff;
tb3 = input[index3] & 0xff;

tr4 = (input[index4] >> 16) & 0xff;
tg4 = (input[index4] >> 8) & 0xff;
tb4 = input[index4] & 0xff;

int tr = (int)(tr1 * coffiecent1 + tr2 * coffiecent4
    + tr3 * coffiecent2 + tr4 * coffiecent3);
int tg = (int)(tg1 * coffiecent1 + tg2 * coffiecent4
    + tg3 * coffiecent2 + tg4 * coffiecent3);
int tb = (int)(tb1 * coffiecent1 + tb2 * coffiecent4
    + tb3 * coffiecent2 + tb4 * coffiecent3);

return new int[]{tr, tg, tb};
}

```

(4) 背景像素颜色

旋转以后的图像比原图像相比，往往宽与高都会增加而产生一些额外背景像素，对此这里采用黑色作为默认背景颜色填充。这部分代码如下：

```

if(row < 0 || row >= height) {
    return new int[]{background.getRed(),
        background.getGreen(), background.getBlue()};
}

if(col < 0 || col >= width) {
    return new int[]{background.getRed(),

```

```
background.getGreen(), background.getBlue());
}
```

其中 background 为 Java 中的 java.awt.Color 类实例。

完整的双线性插值旋转代码可以参考源文件的类 BiLineRotateFilter.java，程序通过参数设置已经实现了 $0^\circ \sim 360^\circ$ 的任意角度旋转，同时还提供了设置不同背景颜色的参数，感兴趣的读者可以进一步修改使用该类。通过 MainUI 运行测试 BiLineRotateFilter 类时，只需要在 ImagePanel 类的 process() 方法中添加如下几行代码即可：

```
BiLineRotateFilter filter = new BiLineRotateFilter();
filter.setDegree(58);
destImage = filter.filter(sourceImage, null);
```

2. 基于错切变换的旋转方法

在旋转角度较小的情况下，旋转矩阵可以通过两次错切变换得到旋转以后的图片，在角度较大的情况下，通过三次错切变换得到旋转以后的图片。角度较小是指旋转角度 θ 小于 15° ，角度较大是指旋转角度小于 90° 。对于旋转角度大于 90° 的，可以先旋转特殊角度，最后错切旋转。旋转矩阵等价于三次错切变换的等式如下：

$$\begin{bmatrix} \cos \alpha & -\sin \alpha \\ \sin \alpha & \cos \alpha \end{bmatrix} = \begin{bmatrix} 1 & -\tan \frac{\alpha}{2} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ \sin \alpha & 1 \end{bmatrix} \begin{bmatrix} 1 & -\tan \frac{\alpha}{2} \\ 0 & 1 \end{bmatrix}$$

其中 α 表示旋转角度。图 7-4 是将一个矩形通过三次错切旋转 45° 的各个步骤实现图。

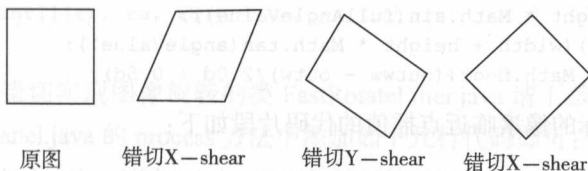


图 7-4 三次错切步骤实现

通过三次错切实现图像旋转，在编码时需要考虑的细节问题有很多，首先由于错切计算采用正切三角函数，其角度的取值范围为 $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$ ，对于角度大于 90° 的旋转，先变换特殊角度（如 90° 、 180° 、 270° ），然后将剩余角度通过三次错切实现。在错切变换过程中，图像的大小会发生形变，计算每次错切变换之后的图像宽与高，以及实际像素匹配问题是一个非常难的编程点。最后采用临近点插值算法实现错切之后各个像素点的像素计算是一个重要的编程技巧。通过三次错切实现图像旋转程序步骤归纳如下：

- 1) 获取源图像像素，宽与高。
- 2) 计算第一次错切 X-shear 之后的图像宽与高，实现 X 方向错切变换。
- 3) 根据第二步结果，计算 Y-shear 错切变换之后的图像宽与高。

- 4) 根据 offsetY 与错切之后图像的实际高度，实现源像素值坐标插值计算。
- 5) 完成 Y-shear，使用临近点插值计算各个像素点值。
- 6) 根据第二次错切 Y-shear 结果，计算最终旋转之后图像的宽与高度。
- 7) 根据 X 方向的 offsetX 与错切之后图像实际宽与高，实现源像素值坐标查找计算。
- 8) 使用临近点插值计算最终旋转之后的图像各个像素值。

程序实现关键代码解析

计算第一次错切 X-shear 之后图像的宽与高的代码片段如下：

```
double angleValue = ((angle/2.0d)/180.0d) * Math.PI;
outh = height;
outw = (int)(width + height * Math.tan(angleValue));
```

计算第二次错切 Y-shear 之后图像的宽与高的代码片段如下：

```
angleValue = ((angle)/180.0d) * Math.PI;
outw = width; // big trick!!!!
outh = (int)(srcWidth * Math.sin(angleValue)
    + height * Math.cos(angleValue));
int outhh = (int)(srcWidth * Math.sin(angleValue) + height);
int offsetY = outhh - outh;
```

计算第三次错切 X-shear 之后图像的宽与高的代码片段如下：

```
angleValue = ((angle/2.0d)/180.0d) * Math.PI;
double fullAngleValue = ((angle)/180.0d) * Math.PI;
outh = height; // big trick
outw = (int)(srcWidth * Math.cos(fullAngleValue)
    + srcHeight * Math.sin(fullAngleValue));
int outww = (int)(width + height * Math.tan(angleValue));
double offsetX = Math.floor((outww - outw)/2.0d + 0.5d);
```

基于像素小数坐标的像素临近点插值的代码片段如下：

```
private int[] getNearestPixels(int[] input, int width, int height,
    double prow, double pcol, boolean yshear) {
    double row = Math.floor(prow);
    double col = Math.floor(pcol);
    if(row < 0 || row >= height) {
        return new int[]{backgroundColor.getRed(),
            backgroundColor.getGreen(),
            backgroundColor.getBlue()};
    }
    if(col < 0 || col >= width) {
        return new int[]{backgroundColor.getRed(),
            backgroundColor.getGreen(),
            backgroundColor.getBlue()};
    }
    double u = yshear ? prow - row : pcol - col;
    int nextRow = (int)(row + 1);
```

```

int nextCol = (int)(col + 1);
if((col + 1) >= width) {
    nextCol = (int)col;
}
if((row + 1) >= height) {
    nextRow = (int)row;
}
int index1 = yshear?(int)(row * width + col) :
    (int)(row * width + col);
int index2 = yshear?(int)(nextRow * width + col):
    (int)(row * width + nextCol);

int tr1, tr2;
int tg1, tg2;
int tb1, tb2;

tr1 = (input[index1] >> 16) & 0xff;
tg1 = (input[index1] >> 8) & 0xff;
tb1 = input[index1] & 0xff;

tr2 = (input[index2] >> 16) & 0xff;
tg2 = (input[index2] >> 8) & 0xff;
tb2 = input[index2] & 0xff;

int tr = (int)(tr1 * (1-u) + tr2 * u);
int tg = (int)(tg1 * (1-u) + tg2 * u);
int tb = (int)(tb1 * (1-u) + tb2 * u);

return new int[]{tr, tg, tb};
}

```

完整的基于三次错切实现图像旋转的类 `FastRotateFilter.java` 请下载阅览, 运行测试该类时, 只需要在 `ImagePanel.java` 的 `process` 方法中添加如下几行代码即可:

```

double angle = 45;
FastRatateFilter filter = new FastRatateFilter();
filter.setAngle(angle);
filter.setBackgroundColor(Color.CYAN);
destImage = filter.filter(sourceImage, null);

```

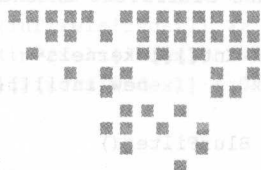
然后运行 `MainUI` 类, 选择一张图片, 单击【处理】按钮即可查看效果。

需要特别说明的是 `FastRotateFilter` 中并没有处理特殊角度(如 90° 、 180° 、 270° 等), 实现任意角度的选择可以在该类的基础上增加对特殊角度旋转支持, 这部分的编程建议读者自己动手实现, 这可帮助你理解所学知识, 获得更多的编程实践。

7.6 小结

本章从介绍图像放大采样入手, 详细介绍了图像快速放大的几种方法, 接着通过深入剖

析图像插值常用的几种方法，由浅入深地学习了临近点插值、双线性插值、双立方插值的数学知识与编程实现。其中对双立方重点剖析了 Blur 与 Sharpen 的不同实现选择。在这些知识的基础上，介绍图像旋转算法，通过对两种不同图像旋转算法的实现，读者很容易总结出它们各自的优缺点，使自己更好地掌握本章知识，活学活用。再次强调一点，源代码是本书内容一部分，请务必动手实现，不断优化本书中已经提供的代码。本章目的是帮助读者解决做项目时经常遇到的关于图像放大、插值、旋转等问题，为读者提供理论知识与实践经验，只要掌握了这些理论知识与方法，就可以在遇到此类问题时游刃有余。同时本章对三角函数、极坐标、简单的矩阵乘法等知识均有所涉及，也希望读者能够花点时间了解一下这些基本的数学知识。



第 8 章

Chapter 8

图像卷积

第 7 章中对像素进行处理时，我们把像素看成一个二维的坐标点，本章换一种角度，从数字信号的角度来考察图像的像素数组，通过数学模型来表达图像像素数组，对图像使用空间域卷积公式进行卷积计算，根据采用的卷积核不同、方向不同、次数不同可以得到不同的图像处理结果，实现一些常见的图像处理。

本章的数学知识中最重要的就是卷积概念与卷积运行，数学上的图像卷积都是连续的，不可分割的，但是对图像来说，卷积可以是离散的，这就为图像卷积运行提供了极大的方便。

希望通过本章的学习，读者能对图像有一个新的认识，不再只是以像素来看待图像，而是可以从信号与能量的角度来看待。思路的转变往往给人带来质的改变，希望本章的学习为读者打开图像处理的一个新通道。

8.1 模糊也是一种美

现在主流的图像处理 APP 都支持照片模糊特效，那种看上去有几分朦胧的美丽给人充分的想象空间，照片也变得有了内涵。其实这种模糊效果，只需要 100 行左右的代码就可以实现，下面就一起来了解一下。

实现处理的编码步骤很简单，只需要读取源图像像素，然后对像素进行适当的处理即可，此时，输出的图片就是具有模糊效果的图片。完整的源代码如下：

```
package com.book.chapter.eight;

import java.awt.image.BufferedImage;
import com.book.chapter.four.AbstractBufferedImageOp;
```



```

        ta = (inPixels[index1] >> 24) & 0xff;
        tr += (inPixels[index1] >> 16) & 0xff;
        tg += (inPixels[index1] >> 8) & 0xff;
        tb += inPixels[index1] & 0xff;
    }
    tr = (int)(tr / total);
    tg = (int)(tg / total);
    tb = (int)(tb / total);
    outPixels[index] = (ta << 24)
    | (tr << 16)
    | (tg << 8) | tb;

    // clean up for next pixel
    tr = 0;
    tg = 0;
    tb = 0;
}

setRGB(dest, 0, 0, width, height, outPixels);
return dest;
}
}

```

执行如下几行代码即可得到模糊效果:

```

BlurFilter filter = new BlurFilter();
destImage = filter.filter(sourceImage, null);

```

从上述代码中大致可以看出实现图像模糊需要一个矩阵乘以一个像素点值及其周围像素点值,很多书中把这个矩阵称为卷积核(Convolution Kernel),而把利用卷积核实现像素计算称为图像卷积,模糊只是图像卷积实现的效果之一。下面将更加详细地介绍图像卷积。

8.2 图像空间域卷积

本节将从数字信号处理的观点来看待图像像素值,一幅数字图像可以用函数 F 表示为:

$$F(m,n) = \sum_{n=0}^N \sum_{m=0}^M p(n,m)$$

其中 N 、 M 表示图像的高与宽, $p(n,m)$ 表示点 (n,m) 的像素值。一幅图像可以抽象地看成一个二维数组,其中每个离散点为像素值,图像空间域卷积的作用就是改变图像空间域频率特征。另外通过观察可以看到,图像是由一系列离散的像素点组成的,所以在介绍卷积概念与相关数学知识时,为了更容易让读者理解卷积,更加贴近编程实践,只介绍离散卷积知识。

首先介绍关于卷积的数学知识,离散卷积的数学公式可以表示为如下形式:

$$f(x) = \sum_{i=0}^{i=n} g(i) \sum_{k=0}^{k=m} C(k)$$

其中 $C(k)$ 代表卷积核（矩阵）， $g(i)$ 代表样本数据， $f(x)$ 代表输出结果。举例如下：假设 $g(i)$ 是一个一维的函数，代表的样本数为 $G = [1, 2, 3, 4, 5, 6, 7, 8, 9]$ ，假设 $C(k)$ 是一个一维的卷积操作数，操作数为 $C = [-1, 0, 1]$ ，则输出结果 $f(x)$ 可以表示为

$$F = [1, 2, 2, 2, 2, 2, 2, 1] \quad // \text{边界数据未处理}$$

可以看出卷积从本质上说是将两个不同数组乘积计算产生一个新的数组的方法。对于二维数组的任意一点，其卷积之后的值通过如下计算公式得到：

$$Y(m, n) = \sum_{k=-a}^a \sum_{l=-b}^b X(m+k, n+l) H(k, l)$$

其中 X 表示图像像素点坐标 (m, n) 的值， H 表示输入的卷积核（二维矩阵数组）， k, l 分别表示卷积核的高与宽。假设图像的高与宽分别为 M, N ，则 m 与 n 的取值范围为 $[0, M-1]$ 与 $[0, N-1]$ ， a 与 b 的值分别为 $a = K/2$ 与 $b = L/2$ ，示意图如图 8-1 所示。

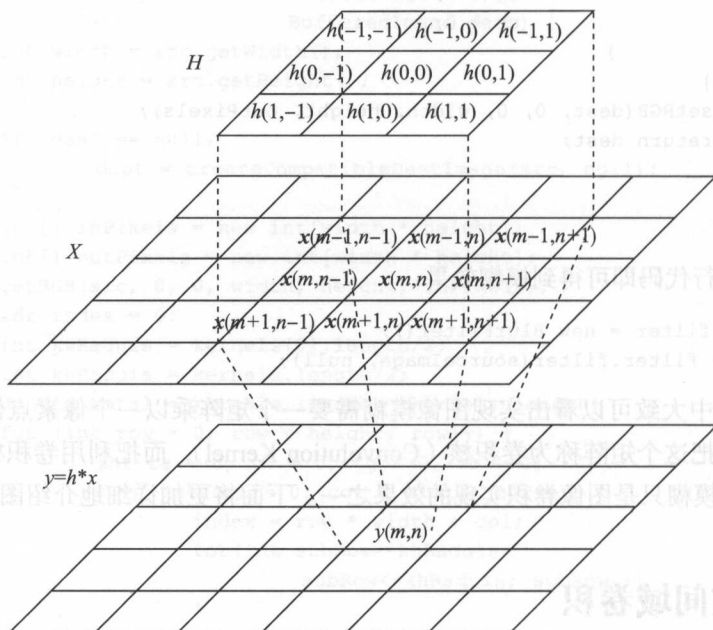


图 8-1 高斯卷积示意图

通常在很多图像相关知识介绍中会提到滤波（Filter），然后给出一个矩阵（Matrix）或卷积核（Convolution kernel），显然这个时候我们知道它是在说卷积，虽然本书不涉及频率域卷积知识的介绍，但还是不得不提及一下，卷积操作不只是可以在空间域完成，还可以对图像进行傅里叶变换，然后到频率域空间完成。本书不会对其进行详细介绍，感兴趣的读者可以自己去学习。

在介绍完卷积基本的数学知识以后，读者不禁要问，为什么需要对图像进行卷积操作？答案是卷积操作可以帮助我们实现图像如下四个方面的处理：

1) 模糊 / 平滑 (Smooth/Blur)

2) 锐化 (Sharpen)

3) 增强 (Intensify)

4) 提高 (Enhance)

本章重点讨论图像卷积的基本知识与图像模糊 / 平滑处理, 其他三种处理将在稍后的章节中介绍。图像完整卷积操作的编程实现步骤大致如下:

1) 读取图像像素数组。

2) 根据二维离散卷积公式, 对每个输入像素进行卷积计算。

3) 对计算结果归一化处理, 输出处理后的像素值。

4) 循环对源像素数组的每个像素进行 2 ~ 3 步的处理, 即可得到处理以后的像素数组。

在处理卷积的过程中, 当卷积核从左到右、从上向下在图像数组上移动时, 如何处理边界像素 (即卷积核与像素数组没有完全重叠之前的图像像素) 也是一个很实际的编程问题, 常见的处理方法有如下三种。

零拓展 (Zero-Extend): 对于没有和像素数组重叠部分的卷积核元素不予考虑, 用零表示。实现代码如下:

```
if (col < 0 || col >= srcWidth
    || row < 0 || row >= srcHeight) {
    continue; // just skip it or make it value as zero
}
```

换行 (Wrap): 通过取模将长度换到下一行, 右边换到左边。实现代码如下:

```
if (col < 0 || col >= srcWidth
    || row < 0 || row >= srcHeight) {
    int nc = col / srcWidth;
    int nr = row / srcHeight;
    row = row - nr * srcHeight; // wrap row
    col = col - nc * srcWidth; // wrap col
    if (row < 0)
        row += srcHeight;
    if (col < 0)
        col += srcWidth;
}
```

剪裁 (Crop): 根据长度靠近边缘大小取值为 0 或图像宽与高, 直接获取边缘像素但是不修改它们的值。实现代码如下:

```
// col
if (col < 0)
    col = 0;
else if (col >= srcWidth)
    col = srcWidth - 1;
else
    col = col;
```

```

// row
if(row < 0)
    row = 0;
else if(row >= srcHeight)
    row = srcHeight - 1;
else
    row = row;

```

上述三种边缘处理方法在图像空间域卷积计算中经常用到，还有一种比较无厘头的处理边缘像素的方法是本书中一直使用的，因为它简单而明了，代码如下：

```

if(row < 0 || row >= height) {
    row = 0;
}
if(col < 0 || col >= width) {
    col = 0;
}

```

一维卷积

对任意一个二维像素数组，分别在 X 与 Y 方向进行一维卷积处理与一次完成二维卷积计算处理得到的结果几乎一样，对图像 X 与 Y 方向分别进行一维卷积的处理步骤大致如下：

- 1) 获取源图像像素数组。
- 2) 对卷积核归一化处理。
- 3) 进行 X 方向的卷积计算，得到输出数组。
- 4) 对 X 方向的卷积计算结果继续进行 Y 方向卷积计算，得到输出数组。
- 5) 对得到的输出像素数组创建返回 Image 对象。

对卷积核归一化预处理的代码如下：

```

// normalization kernels
float sum = 0.0f;
for(int i=0; i<kernels.length; i++)
{
    sum += kernels[i];
}
for(int i=0; i<kernels.length; i++)
{
    kernels[i] = kernels[i]/sum;
}

```

实现图像像素一维单方向卷积的关键代码如下：

```

int cr = kernels2.length/2;
int index = 0;
for(int row=0; row<height; row++)
{
    for(int col=0; col<width; col++)
    {
        float sumr=0, sumg=0, sumb=0;

```

```

for(int nr=-cr; nr<=cr; nr++)
{
    int offsetCol = col + nr;
    if(offsetCol >=0 && offsetCol < width)
    {
        index = row * width + offsetCol;

        // handle edge pixels
        else if(edgeAction == ZERO_EXTEND)
        {
            continue;
        }
        else if(edgeAction == WRAP)
        {
            int ncol = offsetCol / width;
            offsetCol = offsetCol-(ncol * width);
            if(offsetCol < 0)
                offsetCol += width;
        }
        else if(edgeAction == CROP)
        {
            if(offsetCol < 0)
                offsetCol = 0;
            else if(offsetCol >= width)
                offsetCol = width - 1;
        }
        index = row * width + offsetCol;
        int rgb = inPixels[index];
        sumr += kernels2[cr+nr] * ((rgb >> 16) & 0xff);
        sumg += kernels2[cr+nr] * ((rgb >> 8) & 0xff);
        sumb += kernels2[cr+nr] * (rgb & 0xff);
    }
    index = row * width + col;
    outPixels[index] = (255 << 24)
        | (clamp(sumr) << 16)
        | (clamp(sumg) << 8)
        | clamp(sumb);
}
}

```

完整的一维 XY 方向卷积计算代码请参见源文件的 `ConvolutionFilter.java` 类。调用或运行测试该类的代码如下：

```

ConvolutionFilter filter = new ConvolutionFilter();
filter.setKernels(new float[]{1.0f,1.0f,1.0f,1.0f,1.0f});
destImage = filter.filter(sourceImage, null);

```

8.3 盒子模糊与高斯模糊

在学习与理解了卷积数学理论、图像卷积计算基本处理流程、边缘像素处理方法等知识

之后，本节重点介绍一种基于卷积的快速模糊方法——盒子模糊，然后通过基本的高斯数学知识，介绍基于高斯模型的图像卷积模糊。希望通过这些重要知识的学习，读者可加深对图像空间域卷积的理解与掌握，学到利用数学知识解决实际问题的思路与方法。

8.3.1 盒子模糊

从数字信号处理的角度看，图像模糊的本质就是要压制高频信号，保留低频信号。压制高频信号的一个可选择的方法就是卷积滤波。选择一个低频滤波器，对图像上的每个像素实现低频滤波，这样整体效果就是一张数字图像更加模糊，显示更少的细节信息。传统的卷积模糊计算量巨大，程序效率比较低，基于滑动窗口的 Box Blur 是一种快速模糊方法，其结果近似于卷积模糊的结果。

1. 盒子模糊 (Box Blur) 的原理

盒子模糊的数学原理可以分为两部分来解释，首先通过建立查找表，建立索引数组，对卷积像素之和实现按查找表取值，不用额外计算，这样就节约卷积的计算开销，其次在每次卷积核从左向右，或者从上向下移动一个像素距离时，通过计算边缘两个像素的差值实现卷积核移动之后像素和的计算，从而保证一直可以使用查找表来寻找图像像素，而不需要任何乘法计算。最终实现好像一个盒子在图像像素上从左到右，从上到下移动，以较少的开销完成图像的卷积计算实现的功能。

2. 盒子模糊特点

盒子模糊使用的是一种快速图像模糊算法，可以分别在水平与垂直方向单独完成，盒子模糊的效果依赖于以下三个参数值：

- x 方向上盒子模糊半径 H Radius，半径越大 X 方向越模糊。
- y 方向上盒子模糊半径 V Radius，半径越大 Y 方向越模糊。
- X 与 Y 方向盒子模糊的迭代次数，迭代次数越多图像越模糊。

同时，当 X 方向或 Y 方向模糊半径越大时，则在该方向对图像的拉伸效果越明显，在单独 X 方向或 Y 方向盒子模糊时，有移动模糊效果。

3. 关键代码解析

下面根据盒子模糊的基本原理与特征，编程实现盒子模糊算法。首先要根据输入的房子模糊半径建立查找表，根据半径计算查找表大小与定义查找表的代码如下：

```
int tableSize = 2*radius+1;
int divide[] = new int[256*tableSize];
```

初始化建立像素查找表的代码如下：

```
for ( int i = 0; i < 256*tableSize; i++ )
    divide[i] = i/tableSize;
```

根据查找表索引得到每个处理以后像素值，完成 X 方向或 Y 方向上盒子模糊的代码

如下:

```
int inIndex = 0;

// 每一行
for ( int y = 0; y < height; y++ ) {
    int outIndex = y;
    int ta = 0, tr = 0, tg = 0, tb = 0;

    // 初始化盒子里面的像素和
    for ( int i = -radius; i <= radius; i++ ) {
        int rgb = in[inIndex + ImageMath.clamp(i, 0, width-1)];
        ta += (rgb >> 24) & 0xff;
        tr += (rgb >> 16) & 0xff;
        tg += (rgb >> 8) & 0xff;
        tb += rgb & 0xff;
    }

    // 每一列, 每一个像素
    for ( int x = 0; x < width; x++ ) {
        // 赋值到输出像素
        out[ outIndex ] = (divide[ta] << 24)
            | (divide[tr] << 16)
            | (divide[tg] << 8)
            | divide[tb];
        // 移动盒子一个像素距离
        int i1 = x+radius+1;
        // 检测是否达到边缘
        if ( i1 > widthMinus1 )
            i1 = widthMinus1;
        // 将要移出的一个像素
        int i2 = x-radius;
        if ( i2 < 0 )
            i2 = 0;
        int rgb1 = in[inIndex+i1];
        int rgb2 = in[inIndex+i2];
        // 计算移除与移进像素之间的差值, 更新像素和
        ta += ((rgb1 >> 24) & 0xff) - ((rgb2 >> 24) & 0xff);
        tr += ((rgb1 & 0xff0000) - (rgb2 & 0xff0000)) >> 16;
        tg += ((rgb1 & 0xff00) - (rgb2 & 0xff00)) >> 8;
        tb += (rgb1 & 0xff) - (rgb2 & 0xff);
        // 继续到下一行
        outIndex += height;
    }
    // 继续到下一行
    inIndex += width;
}
```

为了让读者更好地理解这段代码, 这里特地加上更多的中文注释来解释每个实现的关键点。

完整的盒子模糊算法 BoxBlurFilter.java 的代码如下：

```
package com.book.chapter.eight;

import java.awt.image.BufferedImage;

import com.book.chapter.four.AbstractBufferedImageOp;
import com.book.image.util.ImageMath;

public class BoxBlurFilter extends AbstractBufferedImageOp {

    private int hRadius;
    private int vRadius;
    private int iterations = 1;

    public BufferedImage filter( BufferedImage src,
        BufferedImage dst ) {
        int width = src.getWidth();
        int height = src.getHeight();

        if ( dst == null )
            dst = createCompatibleDestImage( src, null );

        int[] inPixels = new int[width*height];
        int[] outPixels = new int[width*height];
        getRGB( src, 0, 0, width, height, inPixels );

        for ( int i = 0; i < iterations; i++ ) {
            blur( inPixels, outPixels, width, height, hRadius );
            blur( outPixels, inPixels, height, width, vRadius );
        }

        setRGB( dst, 0, 0, width, height, inPixels );
        return dst;
    }

    public void blur( int[] in, int[] out,
        int width, int height, int radius ) {
        int widthMinus1 = width-1;
        int tableSize = 2*radius+1;
        int lookupTable[] = new int[256*tableSize];

        // 建立查找表
        for ( int i = 0; i < 256*tableSize; i++ )
            lookupTable[i] = i/tableSize;

        int inIndex = 0;

        // 每一行
        for ( int y = 0; y < height; y++ ) {
            int outIndex = y;
```

```

int ta = 0, tr = 0, tg = 0, tb = 0;

// 初始化盒子里面的像素和
for ( int i = -radius; i <= radius; i++ ) {
    int rgb = in[inIndex + ImageMath.clamp(i, 0, width-1)];
    ta += (rgb >> 24) & 0xff;
    tr += (rgb >> 16) & 0xff;
    tg += (rgb >> 8) & 0xff;
    tb += rgb & 0xff;
}

// 每一列，每一个像素
for ( int x = 0; x < width; x++ ) {
    // 赋值到输出像素
    out[ outIndex ] = (lookupTable[ta] << 24)
        | (lookupTable[tr] << 16)
        | (lookupTable[tg] << 8)
        | lookupTable[tb];

    // 移动盒子一个像素距离
    int i1 = x+radius+1;
    // 检测是否达到边缘
    if ( i1 > widthMinus1 )
        i1 = widthMinus1;
    // 将要移出的一个像素
    int i2 = x-radius;
    if ( i2 < 0 )
        i2 = 0;
    int rgb1 = in[inIndex+i1];
    int rgb2 = in[inIndex+i2];
    // 计算移除与移进像素之间的差值，更新像素和
    ta += ((rgb1 >> 24) & 0xff) - ((rgb2 >> 24) & 0xff);
    tr += ((rgb1 & 0xff0000) - (rgb2 & 0xff0000)) >> 16;
    tg += ((rgb1 & 0xff00) - (rgb2 & 0xff00)) >> 8;
    tb += (rgb1 & 0xff) - (rgb2 & 0xff);
    // 继续到下一行
    outIndex += height;
}

// 继续到下一行
inIndex += width;
}

public void setHRadius(int hRadius) {
    this.hRadius = hRadius;
}

public int getHRadius() {
    return hRadius;
}

public void setVRadius(int vRadius) {

```

```

        this.vRadius = vRadius;
    }

    public int getVRadius() { ++radius;
        return vRadius;
    }

    public void setRadius(int radius) {
        this.hRadius = this.vRadius = radius;
    }

    public int getRadius() {
        return hRadius;
    }

    public void setIterations(int iterations){
        this.iterations = iterations;
    }

    public int getIterations() {
        return iterations;
    }

    public String toString() {
        return "Blur/Box Blur...";
    }
}

```

测试盒子模糊的算法时，只需要执行如下几行代码即可：

```

BoxBlurFilter filter = new BoxBlurFilter();
filter.setHRadius(20);
destImage = filter.filter(sourceImage, null);

```

8.3.2 高斯模糊

高斯模糊是一种二维的卷积模糊操作，针对图像完成高斯模糊时，相对于均值模糊来说，其计算量会增加，但是高斯模糊可以实现一些特殊效果，特别是在消除图像噪声（非椒盐噪声）方面，更是有着非常好的效果。一维高斯分布公式如下：

$$G(X) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}}$$

其中 x 的取值范围为 $[-n, n]$ ， σ 表示标准方差。

根据上述公式产生一维高斯卷积核的代码如下：

```

private float[] get1DKernelData(int n, float sigma) {
    float sigma22 = 2*sigma*sigma;
    float Pi2 = 2*(float)Math.PI;
    float sqrtSigmaPi2 = (float)Math.sqrt(Pi2) * sigma ;
    int size = 2*n + 1;
    int index = 0;
}

```

```

float[] kernelData = new float[size];
for(int i=-n; i<=n; i++) {
    float distance = i*i;
    kernelData[index] = (float)Math.exp((-distance)
        /sigma22)/sqrtSigmaPi2;
    index++;
}
return kernelData;

```

假设输入 $n = 1$, $\sigma = 1$ 时, 输出的 Kernel 数据为: 0.24197073, 0.3989423, 0.24197073。得到卷积核之后, 可以分别在 X 方向和 Y 方向进行高斯模糊, 其效果等同与二维高斯模糊的效果。二维高斯分布的数学公式如下:

$$G(x,y)=\frac{1}{2\pi\sigma^2}e^{-\frac{x^2+y^2}{2\sigma^2}}$$

其中 x 、 y 的取值范围决定卷积核大小, σ 表示方差取值范围为正整数, 通常把 $\frac{1}{2\pi\sigma^2}$ 称为归一化因子。根据二维高斯分布公式, 生成二维高斯卷积核的代码如下:

```

public float[][] get2DKernalData(int n, float sigma) {
    int size = 2*n + 1;
    float sigma22 = 2*sigma*sigma;
    float sigma22PI = (float)Math.PI * sigma22;
    float[][] kernelData = new float[size][size];
    int row = 0;
    for(int i=-n; i<=n; i++) {
        int column = 0;
        for(int j=-n; j<=n; j++) {
            float xDistance = i*i;
            float yDistance = j*j;
            kernelData[row][column] = (float)Math.exp(-(xDistance
                + yDistance)/sigma22)/sigma22PI;
            column++;
        }
        row++;
    }
    return kernelData;
}

```

当 $n = 1$, $\sigma = 1$ 时, 对应输出的高斯卷积核矩阵为:

0.058549833	0.09653235	0.058549833
0.09653235	0.15915494	0.09653235
0.058549833	0.09653235	0.058549833

同样 2D 高斯分布的图可以表示为图 8-2。

高斯模糊在图像处理中是一种低通滤波, 会除去图像的细节而保持整体不变化, 在图像美化和特效方面, 高斯模糊有着很多的应用。高斯模糊与均值模糊最大区别在于均值模糊不考虑像素值的权重影响, 而高斯模糊会考虑卷积矩阵各个像素的权重, 从而更好地保留了图像整体特征。完整的高斯模糊算法实现步骤如下:

1) 根据输入参数 n 的大小获取高斯模糊卷积核矩阵数组。

2) 对每个像素计算高斯卷积像素与权重乘积之和。

3) 根据上面第二步的结果除以高斯卷积核总的权重得到输出像素。

4) 循环所有像素即完成图像高斯模糊。

基于二维高斯卷积核的图像模糊关键代码如下：

```
int width = src.getWidth();
int height = src.getHeight();
```

```
if (dest == null)
    dest = createCompatibleDestImage(src, null);
```

```
int[] inPixels = new int[width * height];
int[] outPixels = new int[width * height];
getRGB(src, 0, 0, width, height, inPixels);
int index = 0;
float[][] kernels = get2DKernalData(n, sigma);
int kwRaduis = kernels[0].length/2;
int khRaduis = kernels.length/2;
for (int row = 0; row < height; row++) {
    int ta = 0, tr = 0, tg = 0, tb = 0;
    for (int col = 0; col < width; col++) {
        index = row * width + col;
        double weightSum = 0.0;
        double redSum = 0, greenSum = 0, blueSum = 0;
        for(int subRow=-khRaduis; subRow<=khRaduis; subRow++)
        {
            int nrow = row + subRow;
            if(nrow < 0 || nrow >= height)
            {
                nrow = 0;
            }
            for(int subCol=-kwRaduis; subCol<=kwRaduis; subCol++)
            {
                int ncol = col + subCol;
                if(ncol < 0 || ncol >= width)
                {
                    ncol = 0;
                }
                int index1 = nrow * width + ncol;
                int ta1 = (inPixels[index1] >> 24) & 0xff;
                int tr1 = (inPixels[index1] >> 16) & 0xff;
                int tg1 = (inPixels[index1] >> 8) & 0xff;
                int tb1 = inPixels[index1] & 0xff;
                redSum += tr1 * kernels[subRow + khRaduis][subCol + kwRaduis];
```

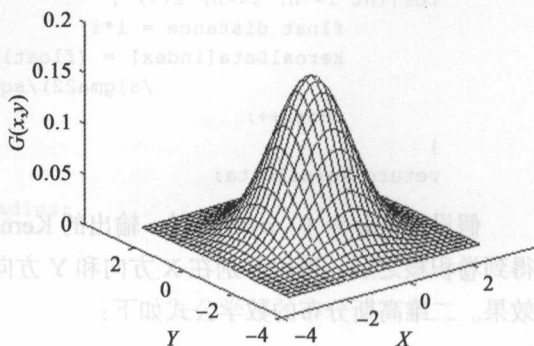


图 8-2 高斯核窗口

```

        greenSum += tgl * kernels[subRow + khRaduis][subCol + kwRaduis];
        blurSum += tbl * kernels[subRow + khRaduis][subCol + kwRaduis];
        weightSum += kernels[subRow + khRaduis][subCol + kwRaduis];
    }
    tr = (int)(redSum / weightSum);
    tg = (int)(greenSum / weightSum);
    tb = (int)(blurSum / weightSum);
    outPixels[index] = (255 << 24) | (tr << 16) | (tg << 8) | tb;

    // clean up for next pixel
    tr = 0;
    tg = 0;
    tb = 0;
}

setRGB(dest, 0, 0, width, height, outPixels);
return dest;

```

完整的高斯模糊算法代码清单请参见源文件的 GaussianBlurFilter.java，测试高斯模糊算法时，只需要执行如下几行代码即可：

```

GaussianBlurFilter filter = new GaussianBlurFilter();
filter.setN(2);
destImage = filter.filter(sourceImage, null);

```

再次强调源代码也是本书的一部分，只有更多的实践才能更好地理解理论知识，加深认知。



注意 高斯模糊与均值模糊最大的区别在于，高斯模糊考虑像素距离权重对计算结果的影响，越靠近中心像素点，像素权重越大，对输出像素值贡献也越大，而均值模糊只是对卷积核与相关所有像素求取算术平均值，所以相比较而言，高斯模糊更好地保留了图像整体特征，而均值模糊则导致像素分辨率降低，细节丢失。

8.4 边缘保留的模糊算法——高斯双边模糊

上节中介绍的两种图像模糊算法都没有很好保留图像细节，特别是边缘细节，而高斯双边模糊算法正是一种边缘保留的图像模糊算法，它的应用也十分广泛，是一种真正实用的图像处理手段，本节将按从原理到编程细节的步骤深入剖析该算法，帮助读者掌握相关的图像处理方法。

1. 高斯双边模糊原理

一般的高斯模糊有如下几个特征：

1) 基于正态分布的线性卷积计算。

2) 权重系数依赖于距离。

3) σ 是经验值, 取决于具体应用场景。

从上面可以看出高斯模糊没有考虑像素变化的影响, 只考虑了像素距离对中心像素的影响, 因此图像边缘也被模糊了。而基于高斯的双边模糊则考虑了像素值变化的影响, 基于高斯分布计算了像素值的权重, 从而产生两个权重表, 一个基于像素空间位置权重。一个基于像素值变化权重。双边模糊的定义如下:

$$BF[I]_p = \frac{1}{W_p} \sum_{q \in S} G_{\sigma_s}(\|p - q\|) G_{\sigma_r}(|I_p - I_q|) I_q$$

其中 $\frac{1}{W_p}$ 称为归一化因子, 是所有权重系数之和; $G_{\sigma_s}(\|p - q\|)$ 是基于空间位置的权重系数的高斯分布; $G_{\sigma_r}(|I_p - I_q|)$ 是基于像素值变化的权重系数的高斯分布; I_q 表示输入像素, 双边滤波定义更形象的表述如图 8-3 所示。

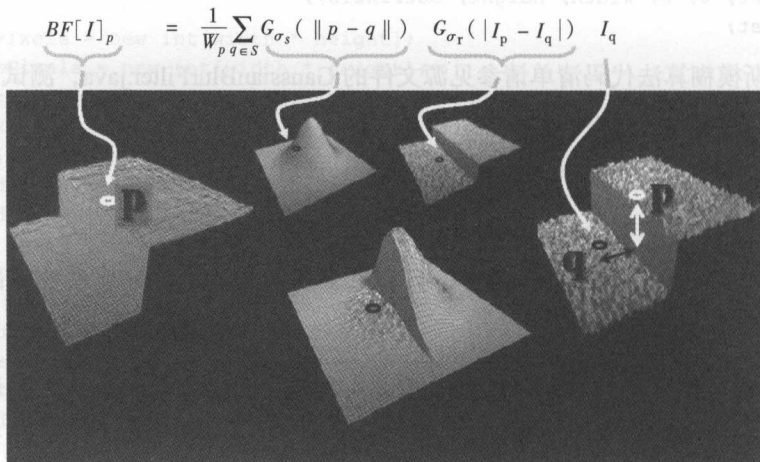


图 8-3 双边模糊示意图

在图 8-3 中, σ_s 指空间位置高斯分布参数, 取值与图像大小成正比, 通常为图像对角线长度的 2%; σ_r 指图像像素高斯分布参数, 取值与图像振幅成正比, 通常为图像梯度的中值或平均值。

前面提到的盒子模糊与高斯模糊本质上都可以看成图像线性滤波, 而双边滤波是非线性滤波卷积, 且计算量比较大, 有时候我们还会对图像完成多次双边滤波即迭代双边滤波, 公式表示为:

$$I_{(n+1)} = BF[I_n]$$

高斯双边模糊的优点有如下几项:

- 1) 边缘保留。
- 2) 虽然是非线性滤波但是为非迭代滤波。
- 3) 计算简单。

相比前面介绍的卷积模糊计算，双边模糊的计算量比较大。

2. 距离计算与查找表建立

计算像素空间位置权重与像素值权重系数时，可以通过预先建立权重查找表来实现，这样减少双边滤波时的计算量，空间距离的计算基于欧几里得距离公式：

$$D(p, q) = \sqrt{\sum_{i=0}^N (p_i - q_i)^2}$$

空间位置的权重表索引建立的代码如下：

```
int size = 2 * radius + 1;
sWeightTable = new double[size][size];
for(int sr = -radius; sr <= radius; sr++) {
    for(int sc = -radius; sc <= radius; sc++) {
        // 计算欧几里得距离
        double delta = Math.sqrt(sr * sr + sc * sc)/sigmas;
        // 根据一维高斯公式，计算高斯权重系数
        double deltaDelta = delta * delta;
        int row = sr + radius;
        int col = sc + radius;
        sWeightTable[row][col] = Math.exp(deltaDelta * factor);
    }
}
```

其中 sWeightTable 为像素位置权重查找表数组，sigmas 为空间位置高斯分布参数。

图像像素高斯分布权重查找表建立的代码如下：

```
// 像素值范围是[0,255]
rWeightTable = new double[256];
// 计算像素值的高斯权重
for(int i=0; i<256; i++) {
    double delta = Math.sqrt(i * i) / sigmar;
    double deltaDelta = delta * delta;
    rWeightTable[i] = Math.exp(deltaDelta * factor);
}
```

其中 rWeightTable 为图像像素高斯分布权重查找表，sigmar 为图像像素高斯分布参数，因为像素取值范围为 0 ~ 255 之间，所以查找表大小为 256。权重系数计算都是基于一维高斯分布公式的。从查找表建立代码可以看出双边模糊（滤波）效果跟下面几个输入参数有关系：

- 1) 空间分布高斯 σ 的大小。
- 2) 像素值分布高斯 σ 的大小。
- 3) 高斯卷积核半径大小。
- 4) 迭代次数（这里不做讨论，当然可以多次重复双边模糊）。

3. 程序实现详解

双边模糊算法的代码实现大致可以分为如下几步：

- 1) 根据输入参数，建立查找表。
- 2) 循环每个像素。
- 3) 对像素进行空间权重与像素值权重高斯卷积计算，同步计算权重和。
- 4) 归一化得到卷积之后的输出像素值。
- 5) 返回计算之后的像素数组。

为了方便大家阅读代码，代码中添加了很多注释，完整的双边模糊代码如下：

```
package com.book.chapter.eight;

import java.awt.image.BufferedImage;
import com.book.chapter.four.AbstractBufferedImageOp;

public class BilateralFilter extends AbstractBufferedImageOp {
    public final static double factor = -0.5d;
    private double sigmas; // space
    private double sigmar; // range
    private int radius;
    private double[][] sWeightTable;
    private double[] rWeightTable;

    public BilateralFilter() {
        radius = 2;
        sigmas = 3;
        sigmar = 30;
    }

    public double getSigmas() {
        return sigmas;
    }

    public void setSigmas(double sigmas) {
        this.sigmas = sigmas;
    }

    public double getSigmar() {
        return sigmar;
    }

    public void setSigmar(double sigmar) {
        this.sigmar = sigmar;
    }

    public int getRadius() {
        return radius;
    }

    public void setRadius(int radius) {
```

```

        this.radius = radius;
    }

    private void buildSpaceWeightTable() {
        int size = 2 * radius + 1;
        sWeightTable = new double[size][size];
        for(int sr = -radius; sr <= radius; sr++) {
            for(int sc = -radius; sc <= radius; sc++) {
                // 计算欧几里得距离
                double delta = Math.sqrt(sr * sr + sc * sc) / sigmas;
                // 根据一维高斯公式, 计算高斯权重系数
                double deltaDelta = delta * delta;
                int row = sr + radius;
                int col = sc + radius;
                sWeightTable[row][col] = Math.exp(deltaDelta * factor);
            }
        }
    }

    private void buildRangeWeightTable() {
        // 像素值范围是[0,255]
        rWeightTable = new double[256];
        // 计算像素值的高斯权重
        for(int i=0; i<256; i++) {
            double delta = Math.sqrt(i * i) / sigmar;
            double deltaDelta = delta * delta;
            rWeightTable[i] = Math.exp(deltaDelta * factor);
        }
    }

    @Override
    public BufferedImage filter(BufferedImage src, BufferedImage dest) {
        int width = src.getWidth();
        int height = src.getHeight();
        //int sigmaMax = (int)Math.max(ds, rs);
        //radius = (int)Math.ceil(2 * sigmaMax);
        radius = (int)Math.max(sigmas, sigmar);
        buildSpaceWeightTable();
        buildRangeWeightTable();
        if (dest == null)
            dest = createCompatibleDestImage( src, null );

        int[] inPixels = new int[width*height];
        int[] outPixels = new int[width*height];
        getRGB( src, 0, 0, width, height, inPixels );
        int index = 0;
        double redSum = 0, greenSum = 0, blueSum = 0;
        double csRedWeight = 0, csGreenWeight = 0, csBlueWeight = 0;
        double csSumRedWeight = 0, csSumGreenWeight = 0, csSumBlueWeight = 0;
        // 由上向下, 从左到右循环每个像素
        for(int row=0; row<height; row++) {

```

```

1) 根据输入 int ta = 0, tr = 0, tg = 0, tb = 0;
2) 循环 for(int col=0; col<width; col++) {
    index = row * width + col;
3) 对像素进行空间 ta = (inPixels[index] >> 24) & 0xff;
4) 归一化得到 tr = (inPixels[index] >> 16) & 0xff;
    tg = (inPixels[index] >> 8) & 0xff;
5) 返回计算之后 tb = inPixels[index] & 0xff;
    int rowOffset = 0, colOffset = 0;
    int index2 = 0;
    int ta2 = 0, tr2 = 0, tg2 = 0, tb2 = 0;
    for(int semirow = -radius; semirow <= radius; semirow++) {
        for(int semicol = -radius; semicol <= radius; semicol++) {
            if((row + semirow) >= 0 && (row + semirow) < height) {
                rowOffset = row + semirow;
            } else {
                rowOffset = 0;
            }
            if((semicol + col) >= 0 && (semicol + col) < width) {
                colOffset = col + semicol;
            } else {
                colOffset = 0;
            }
            index2 = rowOffset * width + colOffset;
            ta2 = (inPixels[index2] >> 24) & 0xff;
            tr2 = (inPixels[index2] >> 16) & 0xff;
            tg2 = (inPixels[index2] >> 8) & 0xff;
            tb2 = inPixels[index2] & 0xff;
            // 在查找表中获取对应权重
            csRedWeight = sWeightTable[semirow+radius][semicol+radius]
                * rWeightTable[(Math.abs(tr2 - tr))];
            csGreenWeight = sWeightTable[semirow+radius][semicol+radius]
                * rWeightTable[(Math.abs(tg2 - tg))];
            csBlueWeight = sWeightTable[semirow+radius][semicol+radius]
                * rWeightTable[(Math.abs(tb2 - tb))];
            // 累加权重之和
            csSumRedWeight += csRedWeight;
            csSumGreenWeight += csGreenWeight;
            csSumBlueWeight += csBlueWeight;
            // 累加权重像素之和
            redSum += (csRedWeight * (double)tr2);
            greenSum += (csGreenWeight * (double)tg2);
            blueSum += (csBlueWeight * (double)tb2);
        }
    }
    // 归一化获取双边滤波之后的像素值
    tr = (int)Math.floor(redSum / csSumRedWeight);
    tg = (int)Math.floor(greenSum / csSumGreenWeight);
    tb = (int)Math.floor(blueSum / csSumBlueWeight);
    outPixels[index] = (ta << 24) | (clamp(tr) << 16) | (clamp(tg)

```

```

        << 8) | clamp(tb);

        // 清空变量，为下一个像素计算做好初始化
        redSum = greenSum = blueSum = 0;
        csRedWeight = csGreenWeight = csBlueWeight = 0;
        csSumRedWeight = csSumGreenWeight = csSumBlueWeight = 0;

    }

    setRGB( dest, 0, 0, width, height, outPixels );
    return dest;
}
}

```

同样测试与运行双边模糊算法 `BilateralFilter.java` 时，只需要如下几行代码即可：

```

BilateralFilter filter = new BilateralFilter();
destImage = filter.filter(sourceImage, null);

```

双边模糊是游戏与图像美化类手机 APP 经常使用的图像处理方法，是一种非常重要的非线性低通滤波方法。如何提高双边模糊算法的执行速度也是一个很大的挑战，最常用的就是通过近似线性卷积计算实现，感兴趣的读者可以进一步阅读与研究。

8.5 像素格特效

在学习了几种图像模糊方法之后，让我们轻松一下，学习一种很有趣而且实用的图像特效——像素格特效。从本质上来说，图像像素格特效是一种图像向下采样，降低图像像素分辨率的方法，该方法本质是把像素数组分为一个个方格，方格大小由参数决定，求取方格范围内像素的平均值，然后将值赋给该方格内的每个像素，这样就完成了图像的像素格特效，最终的效果如图 8-4 所示。

这也是很多图像处理与编辑类 APP 应用的常见功能，编程实现像素格效果的具体步骤如下：



图 8-4 像素格效果图像

- 1) 获取输入像素数组。
- 2) 根据像素格大小，计算每个像素格内像素平均值。
- 3) 根据计算的平均值给各个像素格内的每个像素赋值。
- 4) 循环完成对每个像素格处理，得到输出像素数组。

完整的图像像素格效果的代码如下：

```

package com.book.chapter.eight;

```



```

        ta = (inPixels[index] >> 24) & 0xff;
        sumred += (inPixels[index] >> 16) & 0xff;
        sumgreen += (inPixels[index] >> 8) & 0xff;
        sumblue += inPixels[index] & 0xff;
    }

    // 计算平均值
    index = row * width + col;
    tr = (int)(sumred/total);
    tg = (int)(sumgreen/total);
    tb = (int)(sumblue/total);
    outPixels[index] = (ta << 24) | (tr << 16) | (tg << 8) | tb;
    // 清空计算下一个像素
    sumred = sumgreen = sumblue = 0;
}

setRGB( dest, 0, 0, width, height, outPixels );
return dest;
}
}

```

代码实现简单明了，其中添加相关注释目的是方便读者阅读与理解。本节所学知识权当放松，增强学习知识过程中的趣味性。默认时像素格的大小为 10×10 像素，运行与测试该 `PixellateFilter.java` 的代码如下：

```

PixellateFilter filter = new PixellateFilter();
destImage = filter.filter(sourceImage, null);

```

8.6 卷积应用：图像去噪

本节将展示图像卷积在现实中的应用，卷积除了产生那些美轮美奂的模糊图片之外，另外还被用来降低或消除图像的各种不同噪声，实现图像平滑预处理，为后续图像处理打下基础，这是很多图像处理算法最初的一步。有噪声的图像可以抽象为如下的数学模型：

$$g(x, y) = f(x, y) + \eta(x, y)$$

其中 $f(x, y)$ 表示原图像像素、 $\eta(x, y)$ 表示该像素噪声、 $g(x, y)$ 表示结果噪声图像像素，根据不同噪声模型大致可以分为高斯噪声、指数噪声、椒盐噪声等。

图像去噪时，根据产生的噪声不同，使用的方法也不一样，不管是在频率域，还是在空间域都可以完成图像去噪。本节内容主要针对空间域，利用卷积知识实现图像滤波去噪声，根据选择的卷积核的不同可以实现不同噪声的滤波，常见的空间域噪声卷积处理大致可以分为如下几种。

1. 均值滤波

均值滤波，是卷积处理中最常用的方法，从频率域的角度来看，均值滤波是一种低通滤波

波器，高频信号将会去掉，因此可以消除图像尖锐噪声，实现图像平滑、模糊等功能。理想的均值滤波会用每个像素和它周围的像素计算出来的平均值替换图像中的像素。采样的卷积核 (Kernel) 数据通常是 3×3 的矩阵，如下表示：

1	1	1
1	1	1
1	1	1

卷积核：
黑色粗体1为中心像素，
周围八个像素计算九个
像素的平均值，替换
中心像素值

按从左到右、从上到下的顺序，卷积核经过图像中的每个像素，最终得到处理后的图像。均值滤波可以加上两个参数，即迭代次数与卷积核矩阵大小。在卷积核矩阵大小相同时，迭代次数越多效果就越好；同样，迭代次数相同的情况下，卷积核矩阵长与宽越大，均值滤波的效果就越明显。常见的平均值的计算方法有如下三种：

1) 算术平均值，当采用算术平均值实现均值滤波时，其滤波的数学公式可以表示为如下：

$$f(x, y) = \frac{1}{mn} \sum_{(s, t) \in S_n} g(s, t)$$

其中 m 、 n 分别表示卷积核行与列的大小。

2) 几何平均值，当采用几何平均值实现均值滤波时，其滤波的数学公式可以表示为如下：

$$f(x, y) = \left[\prod_{(s, t) \in S_n} g(s, t) \right]^{\frac{1}{mn}}$$

其中 m 、 n 分别表示卷积核行与列的大小。

3) 调和平均值，当采用调和平均值实现均值滤波时，其滤波的数学公式可以表示为如下：

$$f(x, y) = \frac{mn}{\sum_{(s, t) \in S_n} \frac{1}{g(s, t)}}$$

基于算术平均值的去噪声只是简单的像素模糊，效果并不是很好，而基于几何平均值的去噪声所产生的效果与算术平均值滤波类似，只是在模糊时保留了更多的图像细节，而最后一种调和平均值滤波几乎可明显去除所有的噪声，特别是对高斯噪声效果尤佳。根据卷积核大小计算中心像素均值的代码如下：

```
private int[] calculateMeans(int[][] windowsPixels)
{
    int rows = windowsPixels.length;
    int cols = windowsPixels[0].length;
    int[] rgb = new int[3];
```

```

double total = rows * cols;
double redSum = 0, greenSum = 0, blueSum = 0;
if(this.type == GEOMETRIC_TYPE)
{
    redSum = 1;
    greenSum = 1;
    blueSum = 1;
}
for(int row=0; row<rows; row++)
{
    for(int col=0; col<cols; col++)
    {
        double r = (windowsPixels[row][col] >> 16) & 0xff;
        double g = (windowsPixels[row][col] >> 8) & 0xff;
        double b = windowsPixels[row][col] & 0xff;
        if(this.type == ARITHMETIC_TYPE)
        {
            redSum += r;
            greenSum += g;
            blueSum += b;
        }
        else if(this.type == GEOMETRIC_TYPE)
        {
            redSum = r * redSum;
            greenSum = g * greenSum;
            blueSum = b * blueSum;
        }
        else if(this.type == HARMONIC_TYPE)
        {
            redSum += 1.0d/r;
            greenSum += 1.0d/g;
            blueSum += 1.0d/b;
        }
    }
}
if(this.type == ARITHMETIC_TYPE)
{
    rgb[0] = (int)(redSum/total);
    rgb[1] = (int)(greenSum/total);
    rgb[2] = (int)(blueSum/total);
}
else if(this.type == GEOMETRIC_TYPE)
{
    rgb[0] = (int)Math.pow(redSum, 1.0d/total);
    rgb[1] = (int)Math.pow(greenSum, 1.0d/total);
    rgb[2] = (int)Math.pow(blueSum, 1.0d/total);
}
else if(this.type == HARMONIC_TYPE)
{

```

```

    rgb[0] = (int)(total/redSum);
    rgb[1] = (int)(total/greenSum);
    rgb[2] = (int)(total/blueSum);
}
return rgb;
}

```

使用均值滤波时，首先根据卷积核矩阵大小计算窗口长度，然后计算卷积窗口下所有像素值之和，边缘像素卷积计算时对 X 方向超出图像宽度 width 的或小于 0 的直接取 0，对 Y 方向同样，超出高度或小于 0 的直接取 0。得到所有像素和之后再除以卷积窗口的大小，即得到最终平均像素值。如果是 RGB 彩色图像，对各个通道进行同样的处理即可。完整的均值滤波的代码请参见源文件的 SmoothFilter.java，运行与测试该代码这里不再赘述。

2. 中值滤波

中值滤波也是消除图像噪声最常见的手段之一，特别是消除椒盐噪声，中值滤波的效果比均值滤波更好。中值滤波与均值滤波的唯一不同之处在于，它不是用均值来替换中心每个像素的，而是将周围像素和中心像素排序以后，取中值。一个 3×3 大小的中值滤波可表示如下：

123	125	126	130	140
122	124	126	127	135
118	120	150	125	134
119	115	119	123	133
111	116	110	120	130

周围八个像素值：
115, 119, 120, 123, 124,
125, 126, 127, 150

中值：124

使用中值滤波时，首先根据卷积核矩阵大小计算窗口长度，然后获得卷积核矩阵重叠下的每个像素值，排序之后，获取中值作为中心点像素值。对每个像素点都重复这样的操作就完成了整个图像的中值滤波。

3. 最大最小值滤波

最大最小值滤波也是一种常见的图像统计滤波方法，其基本原理是用最大最小值之差作为卷积核覆盖像素下的中心像素值，对每个像素完成此操作即实现了图像的最大最小值滤波。假设灰度像素值 3×3 的窗口如下：

22	189	48
150	77	158
10	72	210

最大值为：210
最小值为：10

这里，最大最小值滤波操作中心像素值从 77 变成为 200，最大最小值滤波可表示为如下

公式:

$$f(x, y) = \max\{g(s, t)\} - \min\{g(s, t)\}, \text{ 其中 } (s, t) \in S_y$$

最大最小值滤波也是一种很好的边缘检测的滤波方法, 关于边缘检测的更多话题将在第9章中详细阐述。

4. 最大值滤波

最大值滤波是寻找卷积核窗口内像素值最大的像素来替换中心像素, 对图像的每个像素完成此操作即实现了图像的最大值滤波, 假设灰度像素 3×3 的窗口如下:

22	189	48
150	77	158
10	72	210

最大值为: 210

这里, 最大值滤波操作中心像素 77 变成 210, 最大值滤波可表示为如下公式:

$$f(x, y) = \max_{(s, t) \in S_y} \{g(s, t)\}$$

5. 最小值滤波

最小值滤波与图像最大值滤波的操作类似, 唯一不同的是用窗口内最小像素替换中心像素值, 同样假设灰度像素 3×3 的窗口如下:

22	189	48
150	77	158
10	72	210

最小值为: 10

这里, 最小值滤波操作中心像素从 77 变成了 10, 最小值滤波可表示为如下公式:

$$f(x, y) = \min_{(s, t) \in S_y} \{g(s, t)\}$$

最大值与最小值滤波对图像的椒盐噪声有较好效果。

6. 中间点滤波

中间点滤波操作是把卷积窗口内的像素最大值与最小值相加以后取平均值来替换中心像素值, 同样假设灰度像素 3×3 的窗口如下:

22	189	48
150	77	158
10	72	210

最大值为: 210

最小值为: 10

$$110 = (210 + 10) / 2$$

这里, 中间点滤波操作中心像素从 77 变成 110, 中间点滤波可表示为如下公式:

$$f(x, y) = \frac{1}{2} \left[\max_{(s, t) \in S_y} \{g(s, t)\} + \min_{(s, t) \in S_y} \{g(s, t)\} \right]$$

中间点滤波对图像的高斯噪声与均匀噪声有很好的效果。

均值滤波方法属于线性卷积方法，而其他各种滤波去噪声方法属于统计滤波方法。上面已经实现了均值滤波的编码。对于这几种统计滤波的方法，去噪效果还跟卷积窗口的大小有关系，在类 `StatisticsFilter` 中的默认大小为 3×3 ，但是可以根据输入参数进行调整。编程实现统计滤波去噪声的步骤可以分为如下几步：

- 1) 获取输入图像的像素数组。
- 2) 对卷积窗口内的像素排序。
- 3) 根据选择的统计滤波方法，计算出中心像素值。
- 4) 循环每个像素完成第二和第三步的操作。
- 5) 输出滤波以后的数组。

完整的基于卷积统计滤波 `StatisticsFilter.java` 的代码如下：

```
package com.book.chapter.eight;

import java.awt.image.BufferedImage;
import java.util.Arrays;

import com.book.chapter.four.AbstractBufferedImageOp;

public class StatisticsFilter extends AbstractBufferedImageOp {
    public final static int MAX_FILTER = 1;
    public final static int MIN_FILTER = 2;
    public final static int MIN_MAX_FILTER = 4;
    public final static int MEADIAN_FILTER = 8;
    public final static int MID_POINT_FILTER = 16;

    private int kernel_size = 3; // default 3
    private int type = 8; // default mean type

    public StatisticsFilter() {
        {
            System.out.println("Statistics Filter");
        }
    }

    public int getKernelSize() {
        return kernel_size;
    }

    public void setKernelSize(int kernelSize) {
        this.kernel_size = kernelSize;
    }

    public int getType() {
        return type;
    }
}
```

```

public void setType(int type) {
    this.type = type;
}

@Override
public BufferedImage filter(BufferedImage src, BufferedImage dest) {
    int width = src.getWidth();
    int height = src.getHeight();

    if (dest == null)
        dest = createCompatibleDestImage(src, null);

    int[] inPixels = new int[width*height];
    int[] outPixels = new int[width*height];
    getRGB(src, 0, 0, width, height, inPixels);

    int rows2 = kernel_size/2;
    int cols2 = kernel_size/2;
    int index = 0;
    int index2 = 0;
    float total = kernel_size * kernel_size;
    int[][] matrix = new int[3][(int)total];
    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            int count = 0;
            for (int row = -rows2; row <= rows2; row++) {
                int rowoffset = y + row;
                if (rowoffset < 0 || rowoffset >= height)
                    rowoffset = y;
                for (int col = -cols2; col <= cols2; col++) {
                    int coloffset = col + x;
                    if (coloffset < 0 || coloffset >= width)
                        coloffset = x;
                    index2 = rowoffset * width + coloffset;
                    matrix[0][count] = (inPixels[index2] >> 16) & 0xff;
                    matrix[1][count] = (inPixels[index2] >> 8) & 0xff;
                    matrix[2][count] = inPixels[index2] & 0xff;
                    count++;
                }
            }
            // 统计滤波
            int[] rgb = performFilter(matrix);
            int ia = 0xff;
            int ir = rgb[0];

```

```

        int ig = rgb[1];
        int ib = rgb[2];
        outPixels[index++] = (ia << 24) | (ir << 16) | (ig << 8) | ib;
    }
}

// 返回结果
setRGB( dest, 0, 0, width, height, outPixels );
return dest;
}

private int[] performFilter(int[][] matrix) {
    // pick up one filter from here!!!
    int[] rgb = new int[3];
    int[] trs = matrix[0];
    int[] tgs = matrix[1];
    int[] tbs = matrix[2];
    // 默认升序排序
    Arrays.sort(trs);
    Arrays.sort(tgs);
    Arrays.sort(tbs);
    int count = kernel_size * kernel_size;
    // 中值滤波
    if(this.type == MEADIAN_FILTER)
    {
        rgb[0] = trs[count/2];
        rgb[1] = tgs[count/2];
        rgb[2] = tbs[count/2];
    }
    // 最大最小值滤波
    else if(this.type == MIN_MAX_FILTER)
    {
        rgb[0] = trs[count-1] - trs[0];
        rgb[1] = tgs[count-1] - tgs[0];
        rgb[2] = tbs[count-1] - tbs[0];
    }
    // 最大值滤波
    else if(this.type == MAX_FILTER)
    {
        rgb[0] = trs[count-1];
        rgb[1] = tgs[count-1];
        rgb[2] = tbs[count-1];
    }
    // 最小值滤波
    else if(this.type == MIN_FILTER)
    {
        rgb[0] = trs[0];
        rgb[1] = tgs[0];

```

```

        rgb[2] = tbs[0];
    }
    // 中间点滤波
    else if(this.type == MID_POINT_FILTER)
    {
        rgb[0] = (trs[0] + trs[count-1])/2;
        rgb[1] = (tgs[0] + tgs[count-1])/2;
        rgb[2] = (tbs[0] + tbs[count-1])/2;
    }
    return rgb;
}
}

```

运行与测试统计滤波类时，只需要如下几行代码即可：

```

StatisticsFilter filter = new StatisticsFilter();
filter.setType(StatisticsFilter.MID_POINT_FILTER);
destImage = filter.filter(sourceImage, null);

```

本节介绍了基于卷积的图像恢复与质量提升的方法，这些应用都基于一些简单实用的数学知识，想对图像恢复与质量提升有更深了解的读者，可以自己进一步探索。

8.7 图像锐化、拉普拉斯滤波

基于卷积的图像锐化是图像质量提升的常规手段，是图像空间域卷积滤波方法之一。从本质上来说，拉普拉斯（Laplacian）操作是基于二阶导数的图像增强方法，图像经过拉普拉斯滤波操作之后的好处是可以发现更多的图像细节，这通常被称为图像锐化（Image Sharpen）。注意拉普拉斯滤波只是图像锐化的方法之一，但却是最常见与最重要的方法。

1. 拉普拉斯算子

拉普拉斯操作的完成基于卷积图像与卷积核（窗口），完成卷积功能的拉普拉斯卷积核最常见的是 3×3 大小的卷积核，如图 8-5 所示。

0	-1	0	-1	-1	-1
-1	4	-1	-1	8	-1
0	-1	0	-1	-1	-1

图 8-5 所示的是两种最常用的拉普拉斯卷积核。拉普拉斯滤波的数学公式如下：

图 8-5 算子模板

$$L(x, y) = \frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2}$$

即分别求取 X 方向与 Y 方向的二阶导数，对于一幅离散的数字图像像素数组来说：

X 方向二阶导数可以表示为 $\frac{\partial^2 I}{\partial x^2} = 2f(x, y) - f(x+1, y) - f(x-1, y)$

Y 方向二阶导数可以表示为 $\frac{\partial^2 I}{\partial y^2} = 2f(x, y) - f(x, y-1) - f(x, y+1)$

即得到图 8-5 中 3×3 左边第一个常用拉普拉斯卷积核，第二个是第一个的进一步拓展而已。

2. 锐化步骤

基于拉普拉斯操作实现图像锐化的步骤如下：

- 1) 图像拉普拉斯操作，得到细节保留的图像。
- 2) 基于第一步的结果，寻找最小值，减去最小值之后，根据最大值，将图像 Scale 到 0 ~ 255 之间，输出处理以后像素数组。
- 3) 将第二步输出的像素数组与原像素值一一相加，再减去最小值之后，根据最大值归一化图像像素值到 0 ~ 255 之间。
- 4) 输出到处理以后的像素数组。
- 5) 直方图拉伸，让锐化以后的图像看上去跟原图的亮度保持一致。

3. 关键程序解析

像素 Scale 技巧，如何将不在 0 ~ 255 之间的像素变成 0 ~ 255 之间的像素是一个简单的数学问题，只要找到最大最小值之后，根据比例不变的特性，扩展到 0 ~ 255 取值范围之内即可。完整的 Scale 像素值的实现代码如下：

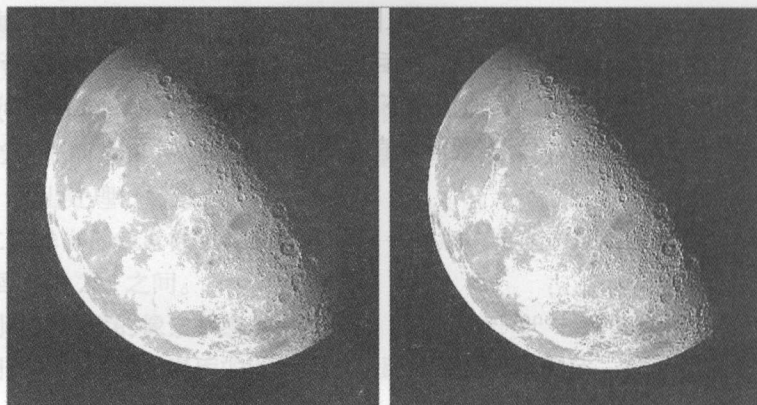
```
private void scalePixels(int[][] rgbPxels)
{
    // scale to 0~255
    float minRed = 255, minGreen = 255, minBlue = 255;
    float maxRed = 0, maxGreen = 0, maxBlue = 0;
    for(int i=0; i<rgbPxels.length; i++)
    {
        minRed = Math.min(minRed, rgbPxels[i][0]);
        minGreen = Math.min(minGreen, rgbPxels[i][1]);
        minBlue = Math.min(minBlue, rgbPxels[i][2]);
    }
    for(int i=0; i<rgbPxels.length; i++)
    {
        rgbPxels[i][0] = (int)(rgbPxels[i][0] - minRed);
        rgbPxels[i][1] = (int)(rgbPxels[i][1] - minGreen);
        rgbPxels[i][2] = (int)(rgbPxels[i][2] - minBlue);
    }
    // filter max value
    for(int i=0; i<rgbPxels.length; i++)
    {
        maxRed = Math.max(maxRed, rgbPxels[i][0]);
        maxGreen = Math.max(maxGreen, rgbPxels[i][1]);
        maxBlue = Math.max(maxBlue, rgbPxels[i][2]);
    }
    for(int i=0; i<rgbPxels.length; i++)
    {
        rgbPxels[i][0] = (int)(rgbPxels[i][0] * (255/maxRed));
        rgbPxels[i][1] = (int)(rgbPxels[i][1] * (255/maxGreen));
        rgbPxels[i][2] = (int)(rgbPxels[i][2] * (255/maxBlue));
    }
}
```

进行直方图像素拉伸操作时，首先要获取图像的直方图，根据直方图选定图像最大与最小值之后，计算取值空间 $s = \max - \min$ ，然后根据像素值 pixel ，如果大于等于 \max 则赋值 255，如果小于等于 \min 则取值 0，其他情况下，根据 $\text{newPixel} = ((\text{pixel} - \min) / s) \times 255$ 公式得到值，此值即为拉伸之后的像素值。实现像素直方图拉伸的代码如下：

```
// 像素的直方图拉伸
int min = 55;
int max = 200;
float dynamic = max - min;
for(int i=0; i<inPixels.length; i++)
{
    if(outPixels[i][0] >= 200)
    {
        outPixels[i][0] = 255;
    }
    else
    {
        outPixels[i][0] = (outPixels[i][0] <= min) ? 0 :
            (int)((outPixels[i][0]-min)/dynamic) * 255.0f);
    }
    if(outPixels[i][1] >= 200)
    {
        outPixels[i][1] = 255;
    }
    else
    {
        outPixels[i][1] = (outPixels[i][1] <= min) ? 0 :
            (int)((outPixels[i][1]-min)/dynamic) * 255.0f);
    }
    if(outPixels[i][2] >= 200)
    {
        outPixels[i][2] = 255;
    }
    else
    {
        outPixels[i][2] = (outPixels[i][2] <= min) ? 0 :
            (int)((outPixels[i][2]-min)/dynamic) * 255.0f);
    }
}
```

基于拉普拉斯算子的图像锐化完整的源代码 `LaplacianSharpenFilter.java` 请下载阅览，测试图片为 `moon.png`，在第 8 章的源代码包中。这里要特别说明的是，最后一步中，直方图拉伸最大与最小值选择，是前面介绍的直方图知识的运用，这里不再赘述。而且在图像锐化时，如果没有必要，这最后一步是可以省去的，这取决于实际项目对处理以后图像的要求。最后看一下基于拉普拉斯图像锐化之后与之前的效果对比，如图 8-6 所示。

很显然，处理之后的图像显示了更多的细节。使用 `LaplacianSharpenFilter` 类只需要如下几行代码即可：



原图

处理之后

图 8-6 效果对比图

```
LaplacianSharpenFilter filter = new LaplacianSharpenFilter();
destImage = filter.filter(sourceImage, null);
```

唯一需要注意的是，不同的图像直方图拉伸的最大最小值是不一样的，要根据实际图像的直方图来确定。

8.8 小结

本章由浅入深地介绍了卷积基本数学知识、基本流程、图像卷积的各种卷积核（算子）的应用，重点介绍了图像模糊算法，基于窗口移动的快速卷积算法，以及边缘保留的图像模糊方法。在学习这些知识的基础上，进一步展开了卷积滤波的概念，介绍了基于统计滤波实现图像去噪声的基本方法与流程，最后介绍了基于拉普拉斯算子实现的图像卷积及其应用来实现图像锐化，提升图像的细节特征。

本章内容同样涉及一些基本的数学知识，其中最重要的是关于卷积的定义，以及离散卷积。其次是导数的概念、一阶与二阶导数在离散点的计算技巧，最后是图像像素值处理问题——如何控制图像像素值在 0 ~ 255 之间。希望读者在学习本章内容的同时，阅读了解更多的相关数学知识，进一步巩固、加深对本章内容的理解。

边缘检测与提取

第8章介绍了卷积及其相关的一些重要应用知识，本章继续基于卷积的应用知识，介绍基于图像空间域边缘检测的基本概念、常用的边缘检测滤波算子、完整基于Canny算法的图像边缘检测与提取算法的实现等。首先从基本概念入手，阐述什么是图像的边缘，图像边缘有哪些明显数学特征，然后介绍如何利用卷积算子实现这些特征的提取与无关噪声的去除。

同样本章也会穿插着介绍一些必要的数学知识，帮助读者厘清边缘提取所需数学知识。希望大家在学习图像处理知识的同时，加强相关数学知识学习，不断提高自己利用数学知识解决图像处理实际问题的能力。同时再次强调一下，本书注重实践，源代码也是本书的一部分，请认真仔细阅读，运行与修改。

9.1 什么是图像的边缘

在探讨关于图像边缘的概念之前，首先来看一幅灰度图像及其对应直方图，如图9-1所示。

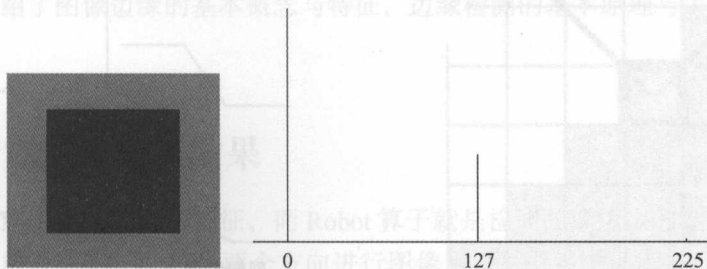


图9-1 边缘直方图

显然从像素值的强度看，值为 0 的像素出现的频率最高，其次是值为 127 的像素。从图像上也可以观察到，从左到右，当颜色从灰度变为黑色时，像素值发生变化，而当从黑色变为灰度时，像素值再次发生变化，其他情况下像素值没有变化，显然那些像素值显著变化的区域就是矩形的边缘。

边缘一般出现在图像两个区域交界处，这些边界处同时也是像素值发生很大变化的地方。一般情况下图像局部区域的像素发生很大变化有如下一些原因：

- ❑ 图像中各种物理物体的边界。
- ❑ 图像中有物理物体的阴影，或者光照与反射等。
- ❑ 各种物理事件。

通常，我们把图像像素发生很大变化的区域称为边缘区域（注意此边缘不是普通理解的边缘），把通过一系列操作得到边缘过程称为边缘提取或边缘检测。

1. 边缘描述与常见边缘模型

如何准确表述图像边缘的特征也是边缘提取中十分重要的一环，通常通过下面几个概念来描述图像边缘特征。

- ❑ 边缘法线：单位向量在该方向上图像像素强度变化最大。
- ❑ 边缘方向：与边缘法线垂直的向量方向。
- ❑ 边缘位置或者中心：图像边缘所在位置。
- ❑ 边缘强度：跟沿法线方向的图像局部对比相关，对比越大，越是边缘。

这里的法线指的是平面几何中所说的过切点的垂直于切线的直线，准确表示如图 9-2 所示。

图像边缘本身依照灰度强度变化可以分为如下几种边缘模型。

❑ 跃迁边缘

跃迁边缘是指图像灰度强度值突然发生很大变化，从一个值突然变化到另外一个相差很大的值，中间没有连续强度值变化。常见的跃迁边缘灰度强度值变化如图 9-3 所示。

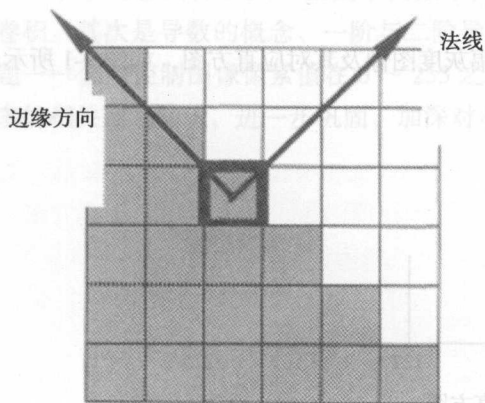


图 9-2 边缘表示

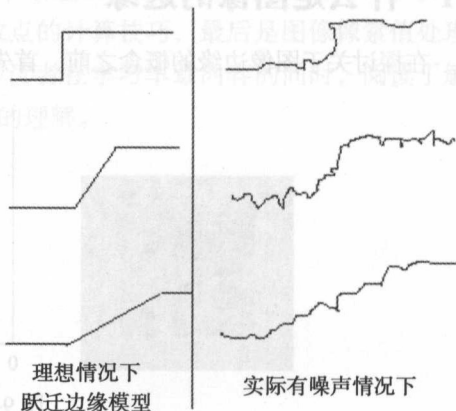


图 9-3 各种边缘类型

□ 斜坡边缘

跟上述的跃迁边缘类似，只是图像灰度强度变化不是突然发生而是一个连续变化的过程。

□ 屋脊边缘

图像像素灰度强度变化突然发生跃迁变化之后，保存该强度一段距离，然后又变化为原来像素的灰度强度。图 9-4 形象地说明了图像屋脊边缘。

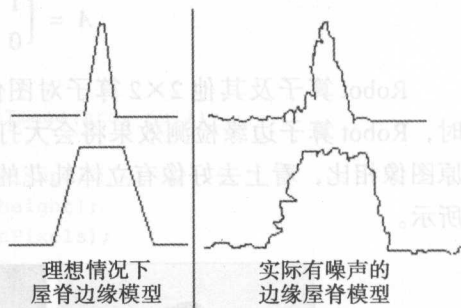


图 9-4 屋脊边缘

2. 边缘检测原理与步骤

在了解了图像边缘各种特征之后，寻找图像边缘就变成如何通过计算寻找那些像素发生变化的区域了，然后通过进一步处理即可得到图像边缘，计算图像像素之间的变化可以通过数学上的一阶导数

来实现，这里所说的图像都是 2D 图像，所以一阶导数分别在两个方向实现，又称为 X 或 Y 方向的一阶偏导数。假设一幅数字图像可以表示为

$$f(i,j) = \sum_{i=0}^N \sum_{j=0}^M P(i,j)$$

其 X 与 Y 方向的一阶偏导数可以表示为：

$$\frac{\partial f}{\partial x} = \lim_{h \rightarrow 0} \frac{f(x+h,y) - f(x,y)}{h} = f(x+1,y) - f(x,y), \text{ 假设 } h=1$$

$$\frac{\partial f}{\partial y} = \lim_{h \rightarrow 0} \frac{f(x,y+h) - f(x,y)}{h} = f(x,y+1) - f(x,y), \text{ 假设 } h=1$$

通过求取图像在 X 与 Y 方向的一阶导数，就可以找到图像的边缘区域，完成对图像的边缘检测，进一步处理以后就可以实现图像边缘提取。通常一个完整的图像边缘检测的流程要包括如下几步：

1) 模糊预处理，目的是降低噪声对图像边缘检测的干扰。

2) 锐化提升，提高图像边缘对比度。

3) 边缘检测，寻找图像边缘区域。

4) 边缘像素定位，目的是从边缘区域中去掉那些非边缘像素。

本节主要介绍了图像边缘的基本概念与特征，边缘检测的基本原理与手段，下面将深入细化这些内容。

9.2 Robot 算子与轧花效果

前面一节介绍了图像边缘的特征，而 Robot 算子就是检测图像边缘的卷积核（算子），而且 Robot 算子是从水平 45° 或 135° 两个方向进行图像边缘的寻找的，Robot 算子对于跃迁边缘有着非常好的效果。Robot 算子的表示如下：

显然从像素值的角度讲， $F_A = f(x, y) - f(x + 1, y + 1)$ 其次是值为 1 的像素值，从图像边缘检测效果来看， $F_B = f(x + 1, y) - f(x, y + 1)$ 其次是值为 1 的像素值。当图像变为灰度时，像素值再次发生变化， $A = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$ 与 $B = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$ 显然那些像素值变化比较大的区域就是图像的边缘。

Robot 算子及其他 2×2 算子对图像噪声非常敏感，换句话说就是当图像有噪声干扰时，Robot 算子边缘检测效果将会大打折扣。基于图像的 Robot 算子 A 或 B 得到的图像与原图像相比，看上去好像有立体轧花的效果。基于 Robot 算子图像边缘的检测效果如图 9-5 所示。

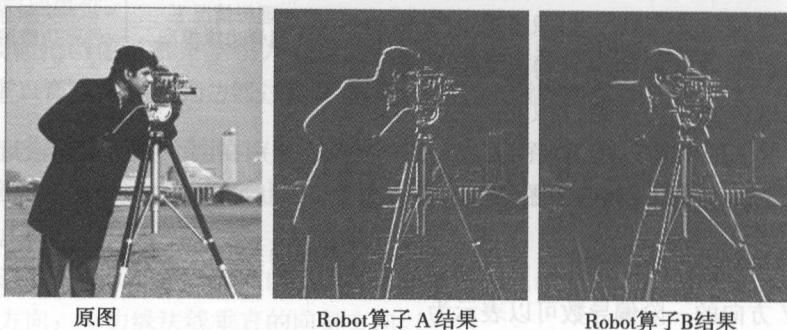


图 9-5 Robot 算子效果

从上述结果可以看出，A 与 B 算子分别对斜上方与斜下方的边缘检测效果比较明显。完整的程序实现 Robot 算子边缘检测的代码如下：

```
package com.book.chapter.nine;

import java.awt.image.BufferedImage;
import com.book.chapter.four.AbstractBufferedImageOp;

public class RobotFilter extends AbstractBufferedImageOp {
    private boolean useA = true;

    public RobotFilter() {
        System.out.println("Robot Filter");
    }

    public boolean isUseA() {
        return useA;
    }

    public void setUseA(boolean useA) {
        this.useA = useA;
    }
}
```



```

@Override
public BufferedImage filter(BufferedImage src, BufferedImage dest) {
    int width = src.getWidth();
    int height = src.getHeight();

    if (dest == null)
        dest = createCompatibleDestImage(src, null);
    //初始化, 获取输入图像像素数组
    int[] inPixels = new int[width * height];
    int[] outPixels = new int[width * height];
    getRGB(src, 0, 0, width, height, inPixels);
    // 每一行、每一列循环每个像素
    int index = 0;
    for (int row = 0; row < height; row++) {
        int ta = 0, tr = 0, tg = 0, tb = 0;
        for (int col = 0; col < width; col++) {
            index = row * width + col;
            // 计算Robot算子, 使用A模板
            if (isUseA())
            {
                int[] rgb1 = getPixel(inPixels, width, height, col, row);

                int[] rgb2 = getPixel(inPixels, width, height, col+1,
                    row+1);
                tr = rgb1[0] - rgb2[0];
                tg = rgb1[1] - rgb2[1];
                tb = rgb1[2] - rgb2[2];
            }
            else
            { // 使用Robot算子B模板
                int[] rgb1 = getPixel(inPixels, width, height,
                    col+1, row);
                int[] rgb2 = getPixel(inPixels, width, height,
                    col, row+1);
                tr = rgb1[0] - rgb2[0];
                tg = rgb1[1] - rgb2[1];
                tb = rgb1[2] - rgb2[2];
            }
            // clamp来处理计算后的结果
            outPixels[index] = (ta << 24) | (clamp(tr) << 16) |
                (clamp(tg) << 8) | clamp(tb);
        }
    }
    setRGB(dest, 0, 0, width, height, outPixels);
    return dest;
}

private int[] getPixel(int[] inPixels, int width, int height, int col,
    int row) {

```



```

        if(col < 0 || col >= width)
            col = 0;
        if(row < 0 || row >= height)
            row = 0;
        int index = row * width + col;
        int tr = (inPixels[index] >> 16) & 0xff;
        int tg = (inPixels[index] >> 8) & 0xff;
        int tb = inPixels[index] & 0xff;
        return new int[] {tr, tg, tb};
    }
}

```

需要特别注意的是，这里用到两个像素值相减，得到的结果可能不在 $0 \sim 255$ 之间，这个时候必须调用 `clamp()` 函数处理计算后的结果。源代码中已经加上了注释，希望读者阅读代码，理解与运行测试。

9.3 Sobel 算子与 Prewitt 算子

Sobel 算子与 Prewitt 算子都是 3×3 的算子，假设有如下的像素块 M ：

a0	a1	a2
a7	[i,j]	a3
a6	a5	a4

则它的 X 与 Y 方向的偏导数计算分别如下：

$$M_x = (a2 + ca3 + a4) - (a0 + ca7 + a6)$$

$$M_y = (a6 + ca5 + a4) - (a0 + ca1 + a2)$$

当系数 $c=1$ 时得到的算子为 Prewitt 算子：

$$M_x = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \quad M_y = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

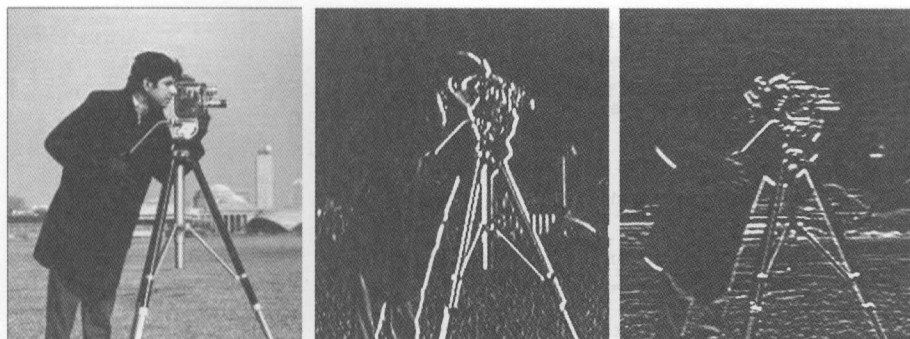
当系数 $c=2$ 时得到的算子为 Sobel 算子：

$$M_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad M_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

基于 Prewitt 算子在 X 与 Y 方向卷积之后得到的图像效果如图 9-6 所示。

基于 Sobel 算子在 X 与 Y 方向卷积之后得到的图像效果如图 9-7 所示。

基于 Sobel 与 Prewitt 算子边缘检测的完整代码如下：

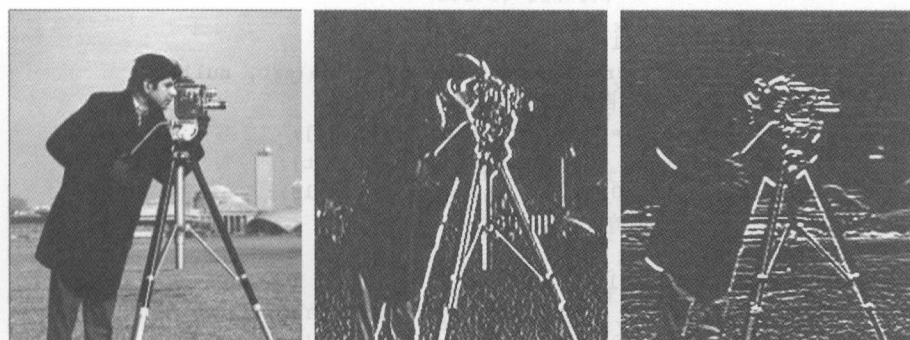


原图

Prewitt算子 X方向边缘检测

Prewitt算子 Y方向边缘检测

图 9-6 Prewitt 算子效果



原图

Sobel算子 X方向边缘检测

Sobel算子 Y方向边缘检测

图 9-7 Sobel 算子效果

```
package com.book.chapter.nine;

import java.awt.image.BufferedImage;
import com.book.chapter.four.AbstractBufferedImageOp;

public class SobolPrewittEdgeDetector extends AbstractBufferedImageOp {
    public final static int SOBEL_TYPE = 1;
    public final static int PREWITT_TYPE = 2;
    public final static int X_DIRECTION = 4;
    public final static int Y_DIRECTION = 8;

    private int type;
    private int direction;
    public SobolPrewittEdgeDetector()
    {
        type = PREWITT_TYPE;
        direction = X_DIRECTION;
    }
    public int getType() {
```

```

        return type;
    }
    public void setType(int type) {
        this.type = type;
    }
    public int getDirection() {
        return direction;
    }
    public void setDirection(int direction) {
        this.direction = direction;
    }
    @Override
    public BufferedImage filter(BufferedImage src, BufferedImage dest) {
        int width = src.getWidth();
        int height = src.getHeight();

        if (dest == null)
            dest = createCompatibleDestImage(src, null);
        //初始化, 获取输入图像像素数组
        int[] inPixels = new int[width * height];
        int[] outPixels = new int[width * height];
        getRGB(src, 0, 0, width, height, inPixels);
        // 每一行、每一列的循环每个像素
        int index = 0;
        // 确定是否为Sobel算子
        int coefficient = (getType() == SOBOLE_TYPE) ? 2 : 1;
        for (int row = 0; row < height; row++) {
            int ta = 0, tr = 0, tg = 0, tb = 0;
            for (int col = 0; col < width; col++) {
                index = row * width + col;
                // X方向边缘检测
                if (getDirection() == X_DIRECTION)
                {
                    int[] a2 = getPixel(inPixels, width, height,
                        col+1, row-1);
                    int[] a3 = getPixel(inPixels, width, height,
                        col+1, row);
                    int[] a4 = getPixel(inPixels, width, height,
                        col+1, row+1);
                    int[] a0 = getPixel(inPixels, width, height,
                        col-1, row-1);
                    int[] a7 = getPixel(inPixels, width, height,
                        col-1, row);
                    int[] a6 = getPixel(inPixels, width, height,
                        col-1, row+1);
                    tr = (a2[0] + coefficient * a3[0] + a4[0]) -
                        (a0[0] + coefficient * a7[0] + a6[0]);
                    tg = (a2[1] + coefficient * a3[1] + a4[1]) -

```

```

        (a0[1] + coefficient * a7[1] + a6[1]);
        tb = (a2[2] + coefficient * a3[2] + a4[2]) -
            (a0[2] + coefficient * a7[2] + a6[2]);
    }
    else
    {
        // Y方向边缘检测
        int[] a6 = getPixel(inPixels, width, height,
            col-1, row+1);
        int[] a5 = getPixel(inPixels, width, height,
            col, row+1);
        int[] a4 = getPixel(inPixels, width, height,
            col+1, row+1);

        int[] a0 = getPixel(inPixels, width, height,
            col-1, row-1);
        int[] a1 = getPixel(inPixels, width, height,
            col, row-1);
        int[] a2 = getPixel(inPixels, width, height,
            col+1, row-1);

        tr = (a6[0] + coefficient*a5[0] + a4[0]) - (a0[
            0]+coefficient*a1[0]+a2[0]);
        tg = (a6[1] + coefficient*a5[1] + a4[1]) - (a0[
            1]+coefficient*a1[1]+a2[1]);
        tb = (a6[2] + coefficient*a5[2] + a4[2]) - (a0[
            2]+coefficient*a1[2]+a2[2]);
    }
    // clamp来处理计算后的结果
    outPixels[index] = (ta << 24) | (clamp(tr) << 16) |
        (clamp(tg) << 8) | clamp(tb);
}
}
setRGB(dest, 0, 0, width, height, outPixels);
return dest;
}

private int[] getPixel(int[] inPixels, int width, int height, int col,
    int row) {
    if(col < 0 || col >= width)
        col = 0;
    if(row < 0 || row >= height)
        row = 0;
    int index = row * width + col;
    int tr = (inPixels[index] >> 16) & 0xff;
    int tg = (inPixels[index] >> 8) & 0xff;
    int tb = inPixels[index] & 0xff;
    return new int[]{tr, tg, tb};
}
}
}

```

其中参数 *type* 选择 Sobel 算子类型或 Prewitt 算子类型，参数 *direction* 决定是进行 X 方向边缘检测还是 Y 方向边缘检测。运行与测试 SobolPrewittEdgeDetector 类时，只需要如下几行代码即可：

```
SobolPrewittEdgeDetector filter = new SobolPrewittEdgeDetector();
filter.setType(SobolPrewittEdgeDetector.SOBOL_TYPE);
filter.setDirection(SobolPrewittEdgeDetector.Y_DIRECTION);
destImage = filter.filter(sourceImage, null);
```

仔细观察运行效果图，发现 X 方向与 Y 方向的边缘恰好形成完整的图像边缘区域，基于 X 与 Y 方向边缘检测结果，如何得到完整图像边缘区域将在下一节中详细讨论。

9.4 图像梯度——大小与角度

前面已经通过基于 X 与 Y 方向的算子得到了图像 X 方向与 Y 方向的一阶偏导数的结果 M_x 与 M_y 。在此基础上，可以得到图像的梯度：

$$\nabla f = \begin{Bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{Bmatrix}$$

其中梯度大小可以表示为： $\text{magn}(\nabla f) = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2} = \sqrt{M_x^2 + M_y^2}$

梯度方向表示为： $\text{dir}(\nabla f) = \tan^{-1}(M_y/M_x)$

根据前面学到的知识， M_x 与 M_y 可以通过 Sobel 或 Prewitt 算子计算得到。进而可以计算得到每个像素的梯度值，其中大小表示边缘的强度，方向表示边缘的方向。很显然，对一幅图像来说，计算得到的梯度大小 $\text{magn}(\nabla f)$ 值只有大于某个阈值 T 时候才可能是边缘。所以通过梯度检测图像边缘的算法，很多时候最后一步就是选取合适的阈值 T 来去掉非边缘像素。完整的基于图像梯度边缘提取算法步骤如下：

- 1) 计算图像 X 方向与 Y 方向的一阶偏导数 M_x 与 M_y 。
- 2) 根据第一步计算结果计算图像梯度。
- 3) 使用阈值 T 细化边缘。

使用阈值细化边缘时，首先需要把图像变成灰度图像，然后选择 $T = 127$ 作为阈值来细化边缘。基于上述步骤实现边缘提取的效果如图 9-8 所示。

根据上述基于图像梯度边缘提取算法的步骤，其中第一步及其代码的实现已经在前面详细论述，第二步计算图像梯度与第三步选择阈值 T 细化边缘在 GradientEdgeFilter 类中实现的完整源代码如下：



图 9-8 运行效果

```

package com.book.chapter.nine;

import java.awt.image.BufferedImage;

public class GradientEdgeFilter extends SobolPrewittEdgeDetector {
    private int threshold = 127;
    public GradientEdgeFilter()
    {
        System.out.println("图像梯度边缘提取法...");
    }

    public int getThreshold() {
        return threshold;
    }

    public void setThreshold(int threshold) {
        this.threshold = threshold;
    }

    @Override
    public BufferedImage filter(BufferedImage src, BufferedImage dest) {
        BufferedImage xImage = super.filter(src, null);
        this.setDirection(Y_DIRECTION);
        BufferedImage yImage = super.filter(src, null);

        int width = src.getWidth();
        int height = src.getHeight();

        if (dest == null)
            dest = createCompatibleDestImage(src, null);
        int[] dxPixels = new int[width * height];
        int[] dyPixels = new int[width * height];
        int[] outPixels = new int[width * height];
        getRGB(xImage, 0, 0, width, height, dxPixels);
        getRGB(yImage, 0, 0, width, height, dyPixels);
    }
}

```



```

其中参数
边缘检测还是
代码即可。
 SobolFruvIt
 filter.setI
 filter.setS
 destImage =

// 计算梯度大小
mred = Math.sqrt(xred * xred + yred * yred);
mgreen = Math.sqrt(xgreen * xgreen + ygreen * ygreen);
mblue = Math.sqrt(xblue * xblue + yblue * yblue);
// 得到图像梯度值
int tr = clamp((int)mred);
int tg = clamp((int)mgreen);
int tb = clamp((int)mblue);
outPixels[index] = (0xff << 24) | (tr << 16) | (tg << 8) | tb;
}

// 根据阈值细化边缘
for (int row = 0; row < height; row++) {
    for (int col = 0; col < width; col++) {
        index = row * width + col;
        int tr = (outPixels[index] >> 16) & 0xff;
        int tg = (outPixels[index] >> 8) & 0xff;
        int tb = outPixels[index] & 0xff;
        tr = tg = tb = (int)(0.299 * (double)tr + 0.587 *
            (double)tg + 0.114 * (double)tb);
        if(tr < threshold || tg < threshold || tb < threshold)
        {
            tr = tg = tb = 0;
        }
        else
        {
            tr = tg = tb = 255;
        }
        outPixels[index] = (0xff << 24) | (tr << 16) | (tg << 8) | tb;
    }
}

setRGB(dest, 0, 0, width, height, outPixels);
return dest;
}

```

关于图像梯度中角度属性的使用将在后面的内容中介绍，本节暂不讨论。本节详细地阐

述了基于图像一阶导数实现图像梯度大小与角度计算完成图像边缘提取的方法,最后一步如何细化边缘有很多种方法,后面的内容将陆续介绍。

9.5 基于二阶导数的图像边缘提取

本节继续学习图像边缘提取方法,不止是基于二阶导数可实现图像梯度的计算与边缘的提取,同样,二阶导数也可以实现图像边缘提取。假设图像边缘可以用函数 $f(x)$ 表示为如图9-9所示的形式。

其对应图像边缘的一阶与二阶导数则如图9-10所示。

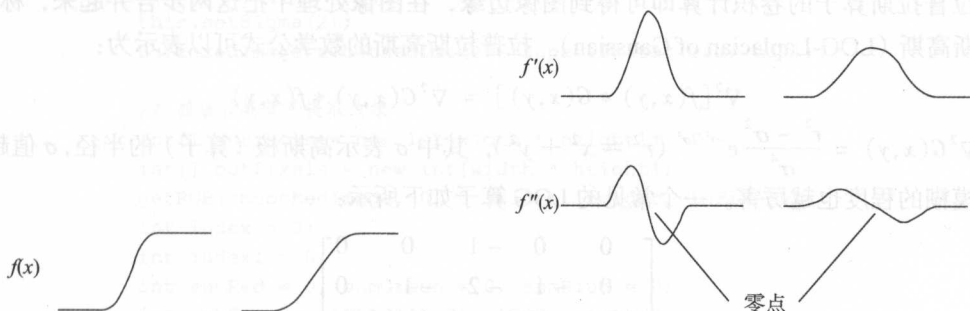


图 9-9 边缘函数

图 9-10 对于一阶与二阶导数结果

图像在 X 方向的二阶导数可以表示为:

$$\frac{\partial^2 f(x,y)}{\partial^2 x} = f''(x,y) = f'(x+1,y) - f'(x,y) = f(x+1,y) - 2f(x,y) + f(x-1,y)$$

对应的卷积算子可以表示为 $[1 \quad -2 \quad 1]$ 。

图像在 Y 方向的二阶导数可以表示为:

$$\frac{\partial^2 f(x,y)}{\partial^2 y} = f''(x,y) = f(x,y+1) - f(x,y) = f(x,y+1) - 2f(x,y) + f(x,y-1)$$

对应的卷积算子可以表示为 $\begin{bmatrix} 1 \\ -2 \\ 1 \end{bmatrix}$

完整的拉普拉斯二阶导数可以表示为: $\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$

其对应的卷积算子表示为:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

基于该算子的几种常见变种的拉普拉斯算子如下:

$$\begin{bmatrix} 1 & -2 & 1 \\ -2 & 4 & -2 \\ 1 & -2 & 1 \end{bmatrix} \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

从上面可以看出，基于二阶导数实现的图像边缘提取只需要一个算子即可完成，比基于一阶导数的边图像梯度边缘提取计算量少，但是二阶导数提取图像边缘不提供边缘方向信息，而且二阶导数边缘提取对图像质量有要求，对有噪声的图像效果不佳。所以通常用二阶导数提取之前，首先会对图像进行低通滤波（模糊），且选择的低通滤波器为高斯滤波器（高斯模糊）。从数字信号处理角度看，图像中的各种细节信息都属于低频信号，而边缘等轮廓属于高频信息，通过高斯模糊就是要消除低频信号对图像高频信号的影响。高斯模糊之后，图像再进行拉普拉斯算子的卷积计算即可得到图像边缘，在图像处理中把这两步合并起来，称为拉普拉斯高斯（LOG-Laplacian of Gaussian）。拉普拉斯高斯的数学公式可以表示为：

$$\nabla^2[f(x,y) * G(x,y)] = \nabla^2 G(x,y) * f(x,y)$$

其中 $\nabla^2 G(x,y) = \frac{r^2 - \sigma^2}{\sigma^4} e^{-r^2/2\sigma^2}$ ($r^2 = x^2 + y^2$)，其中 σ 表示高斯核（算子）的半径， σ 值越大，高斯模糊的程度也越厉害。一个常见的 LOG 算子如下所示：

$$\begin{bmatrix} 0 & 0 & -1 & 0 & 0 \\ 0 & -1 & -2 & -1 & 0 \\ -1 & -2 & 16 & -2 & -1 \\ 0 & -1 & -2 & -1 & 0 \\ 0 & 0 & -1 & 0 & 0 \end{bmatrix}$$

基于二阶导数提取图像边缘的原理部分大致如此，本节中编程实现基于二阶导数的边缘提取完整步骤如下：

- 1) 获取像素数组，进行高斯模糊处理，得到模糊之后的图像。
- 2) 基于拉普拉斯算子完成边缘提取。
- 3) 基于像素最小与最大值完成灰度拉伸。
- 4) 基于阈值完成二值化，得到输出结果。

基于上述步骤实现图像边缘提取效果如图 9-11 所示。



图 9-11 拉普拉斯提取

其中第一步高斯模糊在第 8 章中已经详细解释过，这里不再做赘述，完整的基于拉普拉斯算子的图像二阶导数边缘的提取代码如下：

```
package com.book.chapter.nine;

import java.awt.image.BufferedImage;

import com.book.chapter.eight.GaussianBlurFilter;
```

```

public class LaplacianFilter extends GaussianBlurFilter {
    public final static int[][] LAPLACIAN_OPERATOR = new int[][]{{0,1,0},{1,-
        4,1},{0,1,0}};

    @Override
    public BufferedImage filter(BufferedImage src, BufferedImage dest) {
        int width = src.getWidth();
        int height = src.getHeight();

        if (dest == null)
            dest = createCompatibleDestImage(src, null);

        // 5x5 高斯模糊窗口
        this.setN(2);
        this.setSigma(2);
        BufferedImage smoothedImage = super.filter(src, null);

        // 拉普拉斯算子提取边缘
        int[] inPixels = new int[width * height];
        int[] outPixels = new int[width * height];
        getRGB(smoothedImage, 0, 0, width, height, inPixels);
        int index = 0;
        int index1 = 0;
        int sumRed = 0, sumGreen = 0, sumBlue = 0;
        int subSize = LAPLACIAN_OPERATOR.length/2;
        for (int row = 0; row < height; row++) {
            for (int col = 0; col < width; col++) {
                index = row * width + col;
                for(int subRow=-subSize; subRow<=subSize; subRow++)
                {
                    int nrow = row + subRow;
                    if(nrow < 0 || nrow >= height )
                    {
                        nrow = 0;
                    }
                    for(int subCol=-subSize; subCol<=subSize; subCol++)
                    {
                        int ncol = col + subCol;
                        if(ncol < 0 || ncol >= width)
                        {
                            ncol = 0;
                        }
                        index1 = nrow * width + ncol;
                        int tr = (inPixels[index1] >> 16) & 0xff;
                        int tg = (inPixels[index1] >> 8) & 0xff;
                        int tb = inPixels[index1] & 0xff;
                        // 提取边缘
                        sumRed = sumRed + (tr * LAPLACIAN_OPERATOR[subRow + subSize][subCol + subSize]);
                        sumGreen = sumGreen + (tg * LAPLACIAN_OPERATOR[subRow + subSize][subCol + subSize]);
                    }
                }
            }
        }
    }
}

```

9.6 经典边缘提取算法

本节介绍边缘提取经典算法。边缘提取是图像处理中的一个重要问题，希望读者自己能实现。

- 1) 首先将图像转换为灰度图像。
- 2) 通过高斯模糊卷积实现降噪。
- 3) 计算图像梯度的大小与角度。
- 4) 非最大信号压制。
- 5) 双阈值边缘连接。
- 6) 二值化图像显示。

```

        OPERATOR[subRow + subSize][subCol +
        subSize]);
        sumBlue = sumBlue + (tb * LAPLACIAN_
        OPERATOR[subRow + subSize][subCol +
        subSize]);
    }

    // clamp来处理计算后的结果
    outPixels[index] = (0xff << 24) | (clamp(sumRed) << 16)
    | (clamp(sumGreen) << 8) | clamp(sumBlue);
    // 重置为下个像素计算结果
    sumRed = 0;
    sumGreen = 0;
    sumBlue = 0;
}

// 图像灰度化, 寻找最大最小灰度值
float min = 255, max = 0;
for (int row = 0; row < height; row++) {
    for (int col = 0; col < width; col++) {
        index = row * width + col;
        int tr = (outPixels[index] >> 16) & 0xff;
        int tg = (outPixels[index] >> 8) & 0xff;
        int tb = outPixels[index] & 0xff;
        tr = tg = tb = (int)(0.299 * (double)tr + 0.587 *
        (double)tg + 0.114 * (double)tb);
        min = Math.min(min, tr);
        max = Math.max(max, tr);
        outPixels[index] = (0xff << 24) | (tr << 16) | (tg << 8) | tb;
    }
}

// 灰度拉伸
float scale = max - min;
double sum = 0;
for (int row = 0; row < height; row++) {
    for (int col = 0; col < width; col++) {
        index = row * width + col;
        int gray = (outPixels[index] >> 16) & 0xff;
        if (gray >= max)
        {
            gray = 255;
        }
        else if (gray <= (min+4))// want to remove some noise
        {
            gray = 0;
        }
        else
        {
            gray = (int)((gray-min)*(255.0f/scale));

```

```

    }
    sum += gray;
    outPixels[index] = (0xff << 24) | (gray << 16) | (gray << 8) | gray;
}
}

// 简单的二值化
int means = (int)(sum / outPixels.length);
for(int i=0; i<outPixels.length; i++)
{
    int gray = (outPixels[i] >> 16) & 0xff;
    if(gray <= means)
    {
        gray = 0;
    }
    else
    {
        gray = 255;
    }
    outPixels[i] = (0xff << 24) | (gray << 16) | (gray << 8) | gray;
}
setRGB(dest, 0, 0, width, height, outPixels);
return dest;
}
}

```

本节与前一节中完成边缘提取以后，图像的边缘还是比较宽。解决这个问题有两种方法，一种是通过迭代再次边缘提取来细化边缘，另外一种方法是采用非最大信号压制来实现。下一节中将详细阐述非最大信号压制方法，至于迭代方法多次提取边缘实现细化，希望读者自己实现。

9.6 经典边缘提取算法——Canny Edge Detection

本节介绍边缘提取经典算法 Canny 边缘检测，该算法是在 1986 年由 John F. Canny 开发出来的，Canny 边缘提取算法是一种基于梯度计算边缘的方法，标准的 Canny 边缘提取算法包括如下几步：

- 1) 首先将图像转换为灰度图像。
- 2) 通过高斯模糊卷积实现降噪。
- 3) 计算图像梯度的大小与角度。
- 4) 非最大信号压制。
- 5) 双阈值边缘连接。
- 6) 二值化图像显示。

下面将逐一分析各个步骤的基本操作及其代码实现。

1. 灰度图像转换

在提取边缘之前，对输入的彩色图像进行灰度转换，将图像从 RGB 色彩空间转换到值在 0 ~ 255 之间的灰度值空间中。实现图像像素从 RGB 转换到灰度的公式如下：

$$\text{gray} = R \times 0.299 + G \times 0.587 + B \times 0.114$$

循环每个输入像素获取 RGB 值，将计算得到的结果作为每个像素的输出像素，即完成了图像灰度转换，实现图像灰度转换的代码如下：

```
int[] inPixels = new int[width*height];
int[] outPixels = new int[width*height];
getRGB( src, 0, 0, width, height, inPixels );
int index = 0;
for(int row=0; row<height; row++) {
    int ta = 0, tr = 0, tg = 0, tb = 0;
    for(int col=0; col<width; col++) {
        index = row * width + col;
        ta = (inPixels[index] >> 24) & 0xff;
        tr = (inPixels[index] >> 16) & 0xff;
        tg = (inPixels[index] >> 8) & 0xff;
        tb = inPixels[index] & 0xff;
        int gray= (int)(0.299 *tr + 0.587*tg + 0.114*tb);
        outPixels[index] = (ta << 24) | (gray << 16) | (gray << 8) | gray;
    }
}
```

2. 高斯模糊降噪

Canny 边缘提取算法是基于梯度计算实现的边缘计算方法，对噪声比较敏感，而高斯模糊的目的就是通过图像在空间域低通滤波实现噪声降低，从而减小噪声干扰。该步可以用公式表达如下：

$$S(x, y) = G_{\sigma}(x, y) \otimes f(x, y)$$

其中 $f(x, y)$ 表示原像素点集合、 $S(x, y)$ 表示处理以后的像素集合。 σ 则决定高斯窗口的大小与模糊程度。计算生成高斯卷积核的代码如下：

```
// 计算高斯卷积核
float kernel[][] = new float[gaussianKernelWidth][gaussianKernelWidth];
for(int x=0; x<gaussianKernelWidth; x++)
{
    for(int y=0; y<gaussianKernelWidth; y++)
    {
        kernel[x][y] = gaussian(x, y, gaussianKernelRadius);
    }
}
```

完成像素高斯卷积的代码如下：

```

// 高斯模糊——灰度图像
int krr = (int)gaussianKernelRadius;
for (int row = 0; row < height; row++) {
    for (int col = 0; col < width; col++) {
        index = row * width + col;
        double weightSum = 0.0;
        double redSum = 0;
        for(int subRow=-krr; subRow<=krr; subRow++){
            {
                int nrow = row + subRow;
                if(nrow >= height || nrow < 0)
                {
                    nrow = 0;
                }
                for(int subCol=-krr; subCol<=krr; subCol++){
                    {
                        int ncol = col + subCol;
                        if(ncol >= width || ncol <=0)
                        {
                            ncol = 0;
                        }
                        int index2 = nrow * width + ncol;
                        int tr1 = (inPixels[index2] >> 16) & 0xff;
                        redSum += tr1*kernel[subRow+krr][subCol+krr];
                        weightSum += kernel[subRow+krr][subCol+krr];
                    }
                }
            }
            int gray = (int)(redSum / weightSum);
            outPixels[index] = gray;
        }
    }
}

```

因为是基于灰度像素值的，所以只需要计算灰度值模糊即可。通过高斯模糊降噪以后，像素数据就可以进行下一步处理。

3. 计算图像梯度大小与角度

计算梯度主要基于图像在 X 方向与 Y 方向的一阶偏导数实现，公式表述如下：

$$G_x(x,y) \approx [S(x,y+1) - S(x,y) + S(x+1,y+1) - S(x+1,y)]/2$$

$$G_y(x,y) \approx [S(x,y) - S(x+1,y) + S(x,y+1) - S(x+1,y+1)]/2$$

根据 X 与 Y 方向的梯度可以计算图像该像素点的梯度幅值与角度：

$$G(x,y) = \sqrt{G_x^2(x,y) + G_y^2(x,y)}$$

$$\theta(x,y) = \tan^{-1}(G_y(x,y)/G_x(x,y))$$

由于反三角函数角度值的范围为 $[-\frac{\pi}{2}, \frac{\pi}{2}]$ ，为了便于计算，在角度值上面加上 $\frac{\pi}{2}$ ，从而使角度值范围在 $[0^\circ \sim 180^\circ]$ 之间。计算图像梯度及根据图像梯度计算图像幅值与角度的代码

如下：而将逐一分析各个步骤的基本操作及其代码实现。

```
// 计算梯度-gradient, X方向与Y方向
data = new float[width * height];
magnitudes = new float[width * height];
for (int row = 0; row < height; row++) {
    for (int col = 0; col < width; col++) {
        index = row * width + col;
        // 计算X方向梯度
        float xg = (getPixel(outPixels, width, height, col, row+1) -
                    getPixel(outPixels, width, height, col, row) +
                    getPixel(outPixels, width, height, col+1, row+1) -
                    getPixel(outPixels, width, height, col+1, row))/2.0f;
        float yg = (getPixel(outPixels, width, height, col, row)-
                    getPixel(outPixels, width, height, col+1, row) +
                    getPixel(outPixels, width, height, col, row+1) -
                    getPixel(outPixels, width, height, col+1, row+1))/2.0f;

        // 计算振幅与角度
        data[index] = hypot(xg, yg);
        if(xg == 0)
        {
            if(yg > 0)
            {
                magnitudes[index]=90;
            }
            if(yg < 0)
            {
                magnitudes[index]=-90;
            }
        }
        else if(yg == 0)
        {
            magnitudes[index]=0;
        }
        else
        {
            magnitudes[index] = (float)((Math.atan(yg/xg) * 180)/Math.PI);
        }
        // make it 0 ~ 180
        magnitudes[index] += 90;
    }
}
```

其中 `getPixel()` 方法是获取图像像素值。计算了图像梯度幅值与角度之后，就可以进行下一步的处理了。

4. 非最大信号压制

非最大信号压制的目的是获取细化边缘，其大致思想是根据角度对每个像素幅值比较同方向上两个相邻的像素，如果小于其中任意一个则舍弃，否则保留。其算法大致步骤如下：

- 1) 假设 3×3 的像素区域, 中心像素点为 $P(x, y)$, 定义四个离散边缘角度 0° 、 45° 、 90° 、 135° 。
- 2) 找出中心像素角度与这个四个角度最相邻的角度。
- 3) 根据角度方法, 比较中心像素幅值与相邻两个像素是否为最大, 是则保留, 否则舍弃。

其中离散角度应用如下规则计算:

- 如果角度 θ 的值在 $0 \sim 22.5$ 或 $157.5 \sim 180$ 度之间, 为 0° 。
- 如果角度 θ 的值在 $22.5 \sim 67.5$ 度之间, 为 45° 。
- 如果角度 θ 的值在 $67.5 \sim 112.5$ 度之间, 为 90° 。
- 如果角度 θ 的值在 $112.5 \sim 157.5$ 度之间, 为 135° 。

非最大信号压制的实现代码如下所示:

```
// 非最大信号压制算法 3x3
Arrays.fill(magnitudes, 0);
for (int row = 0; row < height; row++) {
    for (int col = 0; col < width; col++) {
        index = row * width + col;
        float angle = magnitudes[index];
        float m0 = data[index];
        magnitudes[index] = m0;
        if (angle >= 0 && angle < 22.5) // angle 0
        {
            float m1 = getPixel(data, width, height, col-1, row);
            float m2 = getPixel(data, width, height, col+1, row);
            if (m0 < m1 || m0 < m2)
            {
                magnitudes[index] = 0;
            }
        }
        else if (angle >= 22.5 && angle < 67.5) // angle 45
        {
            float m1 = getPixel(data, width, height, col+1, row-1);
            float m2 = getPixel(data, width, height, col-1, row+1);
            if (m0 < m1 || m0 < m2)
            {
                magnitudes[index] = 0;
            }
        }
        else if (angle >= 67.5 && angle < 112.5) // angle 90
        {
            float m1 = getPixel(data, width, height, col, row+1);
            float m2 = getPixel(data, width, height, col, row-1);
            if (m0 < m1 || m0 < m2)
            {
                magnitudes[index] = 0;
            }
        }
        else if (angle >= 112.5 && angle < 157.5) // angle 135 / -45
```

```

        float m1 = getPixel(data, width, height, col-1, row-1);
        float m2 = getPixel(data, width, height, col+1, row+1);
        if(m0 < m1 || m0 < m2)
        {
            magnitudes[index] = 0;
        }
    }
    else if(angle >=157.5) // angle 0
    {
        float m1 = getPixel(data, width, height, col, row+1);
        float m2 = getPixel(data, width, height, col, row-1);
        if(m0 < m1 || m0 < m2)
        {
            magnitudes[index] = 0;
        }
    }
}
}
}

```

5. 双阈值边缘连接

非最大信号压制以后边缘会被细化，但是仍然有一些虽然有幅值但不是边缘的像素被保留，基于一个阈值实现二值化的方法往往导致边缘像素丢失或非边缘像素被保留，Canny 通过设置两个阈值（一个高阈值 TH ，一个低阈值 TL ）来实现边缘像素的连接，通常两个阈值之比为 3:1($TH:TL$)。根据双阈值实现边缘连接时遵守如下规则：

- ❑ 任意小于 TL 的边缘像素被丢弃。
- ❑ 任意大于 TH 的边缘像素被保留。
- ❑ 任意边缘像素 P 为 $TL < P < TH$ ，如果能通过边缘连接到一个边缘大于 TH 而且经过的边缘点都大于最小阈值 TL 的则保留，否则丢弃。

双阈值边缘连接的实现代码如下：

```

// 通常比值为  $TL:TH = 1:3$ ，根据两个阈值完成二值化边缘连接
// 边缘连接——link edges
Arrays.fill(data, 0);
int offset = 0;
for (int row = 0; row < height; row++) {
    for (int col = 0; col < width; col++) {
        if(magnitudes[offset] >= highThreshold && data[offset] == 0)
        {
            follow(col, row, offset, lowThreshold);
        }
        offset++;
    }
}

```

主要通过递归实现边缘连接，其中 follow() 方法实现代码如下所示：

```
private void follow(int x1, int y1, int index, float threshold) {
    int x0 = (x1 == 0) ? x1 : x1 - 1;
    int x2 = (x1 == width - 1) ? x1 : x1 + 1;
    int y0 = y1 == 0 ? y1 : y1 - 1;
    int y2 = y1 == height - 1 ? y1 : y1 + 1;

    data[index] = magnitudes[index];
    for (int x = x0; x <= x2; x++) {
        for (int y = y0; y <= y2; y++) {
            int i2 = x + y * width;
            if ((y != y1 || x != x1)
                && data[i2] == 0
                && magnitudes[i2] >= threshold) {
                follow(x, y, i2, threshold);
                return;
            }
        }
    }
}
```

双阈值处理与边缘连接以后，非边缘像素被丢弃，Canny 边缘提取算法基本完成，但是为了显示图像边缘提取的结果，往往会加上一步二值化来显示 Canny 边缘检测处理之后的结果。

6. 结果二值化显示边缘

基于 Canny 边缘检测的结果，如果值大于 0 则标注为白色，否则为黑色像素，实现代码如下所示：

```
// 二值化显示
for(int i=0; i<inPixels.length; i++)
{
    int gray = clamp((int)data[i]);
    outPixels[i] = gray > 0 ? -1 : 0xff000000;
}
```

最终 CannyEdgeFilter 类运行结果如图 9-12 所示。



图 9-12 Candy 边缘

其中最小阈值 $TL = 10$ ，最大阈值 $TH = 30$ 。完整的 Canny 边缘检测算法源代码请参见源文件的 CannyEdgeFilter.java，读者可以阅读与理解、调用运行该代码。

9.7 小结

本章详细介绍了图像边缘提取的各种常见方法，理论联系实际，对每种方法都做了详尽的原理解释与代码实现，真正做到了用代码实现帮助读者理解各种边缘提取方法与手段。最后介绍了经典的图像边缘检测算法——Canny 边缘检测，一步步解释与实现，帮助读者串联所学知识，融会贯通加深理解，更好地掌握图像边缘提取各种方法之间的区别与联系。同时本章涉及一些基本数学知识的介绍，读者在学习本章知识的同时，了解导数、高斯公式、拉普拉斯公式等数学知识有助于更好地理解与掌握本章内容。

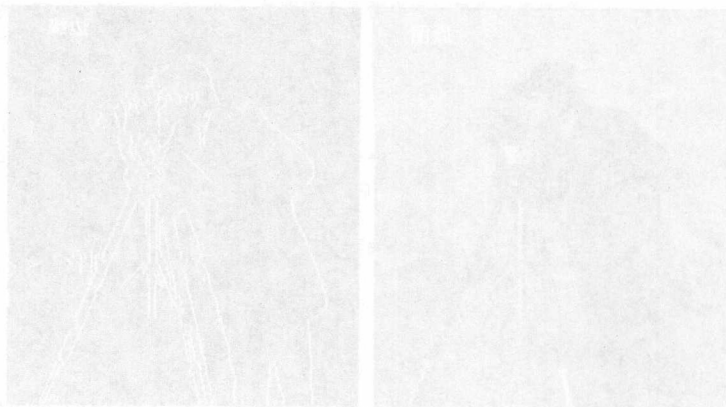


图 9-12 Canny 边缘检测



第10章 Chapter 10

二值图像

第9章学习了图像边缘检测与提取的各种相关知识，多数时候边缘提取都是基于灰度图像开始，最终结果以二值图像出现的。本章将学习二值图像各种分析方法，关于如何将灰度图像转换为二值图像在第8章中已经有详细介绍，本章不再赘述。二值图像分析应用在计算视觉与对象识别与分析等领域是非常重要的图像处理技术，本章将从介绍二值图像的基本特征、半色调显示等基本内容入手，由浅入深系统地介绍二值图像抖动、噪声消除、连通区域寻找与组件标记、边界跟踪、骨架提取、几何特性等常见分析处理方法。这些知识在实际应用中都是非常重要与基础的。同时本章同样也会涉及一些简单数学知识的介绍与应用。希望读者在学习本章知识的同时也多了解一下相关数学知识，这有助于更好地帮助自己学习本章内容。

10.1 二值图像概述与半色调算法

1. 二值图像概述

在具体介绍半色调算法之前，首先介绍二值图像的概念。在图像处理中，二值图像是指只由两个值组成的图像，其中0表示黑色、1表示白色，简单地说，图像只能包含两种颜色——黑色与白色。正是因为二值图像只包含两种颜色，处理起来相对简单方便，所以二值图像分析在很多领域应用中有着非常重要的地位。常见的二值图像分析任务如下：

- 噪声压制或干扰消除
- 连通组件标记与着色
- 轮廓提取与边缘跟踪

- ❑ 距离变换与骨架提取
- ❑ 区域边缘细化
- ❑ 几何特征计算（区域角度、质心）

2. 半色调算法

在打印机功能匮乏的时代，大部分只支持黑白两种颜色，但是很多时候要在报纸上显示灰度图像，解决这一问题的算法称为半色调算法（halftone-algorithm）。从本质来说，半色调算法是一种错误扩散，最常见的基于错误扩散的半色调算法步骤如下。

- 1) 获取图像像素数组 $input[M \times N]$ 。
- 2) 循环每个像素计算：
 - a) 计算当前像素总的错误分布值 EP 。
 - b) 计算当前像素总的错误 $E = P(m, n) + EP$ 。
 - c) 如果 E 大于阈值 T ，则设当前像素为白色，同时 $Eg = 2 \times E - T$ 。如果 E 小于阈值 T ，则设当前像素为黑色，同时 $Eg = E$ 。
 - d) 根据错误扩散矩阵，将 E 分别扩散到其他对应像素。
- 3) 得到输出像素数组 $output[M \times N]$ 。

完整的半色调错误扩散的代码如下：

```
package com.book.chapter.ten;

import java.awt.image.BufferedImage;

import com.book.chapter.four.AbstractBufferedImageOp;

public class HalftoneFilter extends AbstractBufferedImageOp {
    private float[][] error_dist = new float[][]{{0,0.2f,0},{0.6f,0.1f,0.1f}};
    private float threshold;

    public HalftoneFilter()
    {
        threshold = 128;
    }

    @Override
    public BufferedImage filter(BufferedImage src, BufferedImage dest) {
        int width = src.getWidth();
        int height = src.getHeight();
        if (dest == null)
            dest = createCompatibleDestImage(src, null);
        //初始化，获取输入图像像素数组
        int[] inPixels = new int[width * height];
        int[] outPixels = new int[width * height];
        getRGB(src, 0, 0, width, height, inPixels);
```

```

int drow = error_dist.length;
int dcol = error_dist[0].length;
int index = 0;
float eg = 0; // 总的错误
float ep = 0; // 转移到下个像素点的错误分散
for (int row = 0; row < height; row++) {
    for (int col = 0; col < width; col++) {
        index = row * width + col;
        int r1 = (inPixels[index] >> 16) & 0xff;
        float tp = r1 + ep;
        if (tp > threshold) {
            outPixels[index] = -1; // 白色
            eg = tp - 2*threshold;
        }
        else {
            outPixels[index] = 0xff000000; // 黑色
            eg = threshold;
        }
        // 错误扩散功能
        for (int sr=0; sr<drow; sr++) {
            int nrow = sr + row;
            if (nrow >= height) {
                nrow = 0;
            }
            for (int sc=0; sc<dcol; sc++) {
                int ncol = sc + col;
                if (ncol >= width) {
                    ncol = 0;
                }
                int p = getPixel(inPixels, width, height, ncol, nrow);
                p = (int)(p + eg * error_dist[sr][sc]);
                setPixel(inPixels, width, height, ncol, nrow, p);
            }
        }
        setRGB(dest, 0, 0, width, height, outPixels);
        return dest;
    }
}

private void setPixel(int[] input, int width, int height, int col, int row, int p)
{
    if (col < 0 || col >= width)

```

```

        col = 0;
        if(row < 0 || row >= height)
            row = 0;
        int index = row * width + col;
        input[index] = (0xff << 24) | (clamp(p) << 16) | (clamp(p) << 8) | clamp(p);
    }

    private int getPixel(int[] input, int width, int height, int col,
        int row) {
        if(col < 0 || col >= width)
            col = 0;
        if(row < 0 || row >= height)
            row = 0;
        int index = row * width + col;
        int tr = (input[index] >> 16) & 0xff;
        return tr;
    }
}

```

运行与测试 HalftoneFilter.java 时, 只需要在 ImagePanel 的 process 方法中添加如下几行代码即可:

```

HalftoneFilter filter = new HalftoneFilter();
destImage = filter.filter(sourceImage, null);

```

ImagePanel.java 的完整源代码清单请参见源文件中第 3 章的代码。对基于灰度图像的半调色算法, 读者还可以尝试其他不同的错误扩散矩阵, 通过实践加深认识。对于提到的二值图像来说, 一样可以应用半调色算法, 这些内容将在下一节内容中详细讲述。

10.2 图像抖动算法

图像抖动算法本质上是图像半调色算法的一个分支发展, 最常见各种图像抖动算法主要用来显示二值图像, 常见的为基于阈值的抖动方法, 就是大家所说的大于 127 为白色, 反之则为黑色。这里介绍两种基于错误扩散的图像抖动算法, 分别为弗洛伊德·斯坦德伯格抖动 (Floyd-Steinberg Dither) 算法、阿特金森抖动 (Atkinson Dither) 算法。该类算法的大致思想都是获取灰度图像的像素值, 根据像素值查找颜色匹配表, 找到与之颜色相近的色彩之后, 把该颜色值作为该像素点的值, 然后计算两个值之间的差值作为总的错误, 并根据错误扩散矩阵与系数分别把错误加到指定像素点像素值上, 对图像的每个像素点都完成此操作就实现了图像的二值抖动算法。对彩色图像进行处理时, 与灰度图像的处理方法类似, 感兴趣的读者可以自己进一步研究与尝试。

1. 基于错误扩散抖动算法基本步骤

具体步骤如下:

- 1) 获取输入的灰度图像像素值数组。
- 2) 对每个像素值完成以下操作。
 - a) 计算当前像素值 $P(x, y) = OP$ 与给定的颜色查找表中哪个颜色最相近, 得到该颜色值 C 。
 - b) 把 C 当做该像素点 $P(x, y)$ 的像素值, 同时计算 $E = OP - C$ 做该像素点的总错误。
 - c) 根据错误扩散公式, 分别乘上对应系数, 将错误值分配到各个像素源像素点。
- 3) 循环对每个像素完成第二步的操作即完成该算法。

其中弗洛伊德·斯坦德伯格抖动的错误扩散公式如下:

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & * & 0.4375 \\ 0.1875 & 0.3125 & 0.0625 \end{bmatrix}$$

阿特金森抖动的错误扩散公式如下:

$$\begin{bmatrix} 0 & * & 0.125 & 0.125 \\ 0.125 & 0.125 & 0.125 & 0 \\ 0 & 0.125 & 0 & 0 \end{bmatrix}$$

这里的 * 表示当前像素位置, 弗洛伊德·斯坦德伯格抖动在 1976 年由 Robert W. Floyd 和 Louis Steinberg 两个人共同开发, 而阿特金森抖动算法是苹果公司的一个程序员开发出来的, 此人的英文名字就叫 Atkinson。

2. 代码实现

根据上面抖动算法的基本流程可知, 实现弗洛伊德·斯坦德伯格抖动与阿特金森抖动算法最主要的是根据它们各自的扩散矩阵分配错误值到相关的像素点中。其中弗洛伊德·斯坦德伯格抖动的扩散矩阵中有四个非零值, 所以需要将错误值扩散分配到这四个相邻像素中, 而阿特金森抖动的扩散矩阵中有六个非零值, 同样需要将得到的当前像素错误值分配到这六个相关像素点中。完整的抖动二值图像源代码如下:

```
package com.book.chapter.ten;

import java.awt.image.BufferedImage;

import com.book.chapter.four.AbstractBufferedImageOp;

public class BinaryDitherFilter extends AbstractBufferedImageOp {
    public final static int[][] COLOR_PALETTE = new int[][] {{0, 0, 0}, {255, 255, 255}};
    public final static int FLOYD_STEINBERG_DITHER = 1;
    public final static int ATKINSON_DITHER = 2;
    private int method;

    public BinaryDitherFilter()
    {
    }
```



```

        method = FLOYD_STEINBERG_DITHER;
    }

    public int getMethod() {
        return method;
    }

    public void setMethod(int method) {
        this.method = method;
    }

    @Override
    public BufferedImage filter(BufferedImage src, BufferedImage dest) {
        int width = src.getWidth();
        int height = src.getHeight();

        if (dest == null)
            dest = createCompatibleDestImage(src, null);

        // 初始化, 获取输入图像像素数组
        int[] inPixels = new int[width * height];
        int[] outPixels = new int[width * height];
        getRGB(src, 0, 0, width, height, inPixels);
        int index = 0;
        for (int row = 0; row < height; row++) {
            for (int col = 0; col < width; col++) {
                index = row * width + col;
                index = row * width + col;
                int r1 = (inPixels[index] >> 16) & 0xff;
                int g1 = (inPixels[index] >> 8) & 0xff;
                int b1 = inPixels[index] & 0xff;
                int cIndex = getCloseColor(r1, g1, b1);
                outPixels[index] = (255 << 24) | (COLOR_PALETTE[cIndex][0] << 16) |
                    (COLOR_PALETTE[cIndex][1] << 8) | COLOR_PALETTE[cIndex][2];
                // 获取错误
                int[] ergb = new int[3];
                ergb[0] = r1 - COLOR_PALETTE[cIndex][0];
                ergb[1] = g1 - COLOR_PALETTE[cIndex][1];
                ergb[2] = b1 - COLOR_PALETTE[cIndex][2];

                // 错误扩散功能
                if (method == FLOYD_STEINBERG_DITHER)
                {
                    float e1=7f/16f;
                    float e2=5f/16f;
                    float e3=3f/16f;
                    float e4=1f/16f;
                    int[] rgb1 = getPixel(inPixels, width, height, col+1, row, e1, ergb);
                    int[] rgb2 = getPixel(inPixels, width, height, col, row+1, e2, ergb);
                    int[] rgb3 = getPixel(inPixels, width, height, col-1, row+1, e3, ergb);
                    int[] rgb4 = getPixel(inPixels, width, height, col+1, row+1, e4, ergb);
                    setPixel(inPixels, width, height, col+1, row, rgb1);

```

```

        setPixel(inPixels, width, height, col, row+1, rgb2);
        setPixel(inPixels, width, height, col-1, row+1, rgb3);
        setPixel(inPixels, width, height, col+1, row+1, rgb4);
    }
    else if(method == ATKINSON_DITHER)
    {
        float e1=0.125f;
        int[] rgb1 = getPixel(inPixels, width, height, col+1, row, e1, ergb);
        int[] rgb2 = getPixel(inPixels, width, height, col+2, row, e1, ergb);
        int[] rgb3 = getPixel(inPixels, width, height, col-1, row+1, e1, ergb);
        int[] rgb4 = getPixel(inPixels, width, height, col, row+1, e1, ergb);
        int[] rgb5 = getPixel(inPixels, width, height, col+1, row+1, e1, ergb);
        int[] rgb6 = getPixel(inPixels, width, height, col, row+2, e1, ergb);
        setPixel(inPixels, width, height, col+1, row, rgb1);
        setPixel(inPixels, width, height, col+2, row, rgb2);
        setPixel(inPixels, width, height, col-1, row+1, rgb3);
        setPixel(inPixels, width, height, col, row+1, rgb4);
        setPixel(inPixels, width, height, col+1, row+1, rgb5);
        setPixel(inPixels, width, height, col, row+2, rgb6);
    }
    else
    {
        throw new java.lang.IllegalArgumentException("Not Supported
        Dither Mothed!!");
    }
}

setRGB(dest, 0, 0, width, height, outPixels);
return dest;
}

private int getCloseColor(int tr, int tg, int tb) {
    int minDistanceSquared = 255*255 + 255*255 + 255*255 + 1;
    int bestIndex = 0;
    for(int i=0; i<COLOR_PALETTE.length; i++) {
        int rdiff = tr - COLOR_PALETTE[i][0];
        int gdifff = tg - COLOR_PALETTE[i][1];
        int bdiff = tb - COLOR_PALETTE[i][2];
        int distanceSquared = rdiff*rdiff + gdifff*gdifff + bdiff*bdiff;
        if(distanceSquared < minDistanceSquared) {
            minDistanceSquared = distanceSquared;
            bestIndex = i;
        }
    }
    return bestIndex;
}

private void setPixel(int[] input, int width, int height, int col, int row, int[] p)
{

```

```

        if(col < 0 || col >= width)
            col = 0;
        if(row < 0 || row >= height)
            row = 0;
        int index = row * width + col;
        input[index] = (0xff << 24) | (clamp(p[0]) << 16) | (clamp(p[1]) << 8) | clamp(p[2]);
    }

    private int[] getPixel(int[] input, int width, int height, int col,
        int row, float error, int[] ergb) {
        if(col < 0 || col >= width)
            col = 0;
        if(row < 0 || row >= height)
            row = 0;
        int index = row * width + col;
        int tr = (input[index] >> 16) & 0xff;
        int tg = (input[index] >> 8) & 0xff;
        int tb = input[index] & 0xff;
        tr = (int)(tr + error * ergb[0]);
        tg = (int)(tg + error * ergb[1]);
        tb = (int)(tb + error * ergb[2]);
        return new int[]{tr, tg, tb};
    }
}

```

运行与测试这两个抖动算法时，只需要在 `ImagePanel` 类的 `process` 方法中如下几行代码即可：

```

BinaryDitherFilter filter = new BinaryDitherFilter();
filter.setMethod(BinaryDitherFilter.ATKINSON_DITHER);
destImage = filter.filter(sourceImage, null);

```

完整的 `ImagePanel.java` 源代码请参见源文件中第 3 章的代码。基于错误扩散的抖动算法不止本节介绍的这两种，感兴趣的读者可以自己进一步学习相关知识。

10.3 二值图像泛洪填充算法

二值图像几何形状颜色填充标识是在二值图像处理中经常需要用的，其中最常用的算法是泛洪填充算法，该算法有两种实现方法，一种基于递归实现，另外一种基于扫描线实现。其基本思想就是从封闭区域内任意一个像素点开始，填充周围四邻域像素点或八邻域像素点，不断迭代直到封闭区域内的所有像素都被指定的颜色填充为止。另外一种是以某一点为基点，在垂直方向开始扫描，然后对水平方向进行扫描，直到封闭区域内的所有点被填充为止。

1. 基于邻域像素寻找的实现

基于四邻域像素寻找时，假设对开始像素点 $P(x, y)$ 填充了指定的颜色，现在要寻找该像

素点上下左右四个像素点, 如果没有填充, 则填充它们, 并且继续寻找它们的四邻域像素, 直到封闭区域内的所有像素点都被指定颜色填充为止。像素点 P 及其四邻域像素如图 10-1 所示。

基于四邻域填充方法的递归实现代码如下:

```
public void floodFill4(int x, int y, int newColor, int
    oldColor)
{
    if(x >= 0 && x < width && y >= 0 && y < height
        && getColor(x, y) == oldColor && getColor(x,
y) != newColor)
    {
        setColor(x, y, newColor); //set color before starting recursion
        floodFill4(x + 1, y, newColor, oldColor);
        floodFill4(x - 1, y, newColor, oldColor);
        floodFill4(x, y + 1, newColor, oldColor);
        floodFill4(x, y - 1, newColor, oldColor);
    }
}
```

	$P(x, y-1)$	
$P(x-1, y)$	$P(x, y)$	$P(x+1, y)$
	$P(x, y+1)$	

图 10-1 四邻域像素示意图

八邻域泛洪填充算法与四邻域泛洪填充类似, 唯一不同的是它会寻找像素点 $P(x, y)$ 周围的八个像素点继续填充, 直到封闭区域被指定颜色完成填充为止。像素 P 的八邻域像素图如图 10-2 所示。

递归实现八邻域寻找泛洪填充方法的代码如下:

```
public void floodFill8(int x, int y, int newColor,
    int oldColor)
{
    while(getColor(x, y) == oldColor &&
        if(x >= 0 && x < width && y >= 0 && y < height &&
            getColor(x, y) == oldColor && getColor(x, y) != newColor)
        {
            setColor(x, y, newColor); //set color before starting recursion
            floodFill8(x + 1, y, newColor, oldColor);
            floodFill8(x - 1, y, newColor, oldColor);
            floodFill8(x, y + 1, newColor, oldColor);
            floodFill8(x, y - 1, newColor, oldColor);
            floodFill8(x + 1, y + 1, newColor, oldColor);
            floodFill8(x - 1, y - 1, newColor, oldColor);
            floodFill8(x - 1, y + 1, newColor, oldColor);
            floodFill8(x + 1, y - 1, newColor, oldColor);
        }
}
```

$P(x-1, y-1)$	$P(x, y-1)$	$P(x+1, y-1)$
$P(x-1, y)$	$P(x, y)$	$P(x+1, y)$
$P(x-1, y+1)$	$P(x, y+1)$	$P(x+1, y+1)$

图 10-2 八邻域示意图

2. 基于扫描线实现

基于扫描线的泛洪填充算法实现有两种方式, 一种是基于递归方式, 另外一种是基于栈的非递归方式, 两种实现方式虽不同, 但是其基本思想是一样的, 都是对指定的像素点 $P(x, y)$

首先沿着 Y 方向分别在上下扫描填充颜色，然后沿 X 方向分别左右移动一个像素，继续沿 Y 方向的扫描。基于递归实现扫描线算法的代码如下：

```
public void floodFillScanLine(int x, int y, int newColor, int oldColor)
```

```
{
```

```
    if(oldColor == newColor) return;
```

```
    if(getColor(x, y) != oldColor) return;
```

```
    int y1;
```

```
    //draw current scanline from start position to the top
```

```
    y1 = y;
```

```
    while(y1 < height && getColor(x, y1) == oldColor)
```

```
    {
```

```
        setColor(x, y1, newColor);
```

```
        y1++;
```

```
    }
```

```
    //draw current scanline from start position to the bottom
```

```
    y1 = y - 1;
```

```
    while(y1 >= 0 && getColor(x, y1) == oldColor)
```

```
    {
```

```
        setColor(x, y1, newColor);
```

```
        y1--;
```

```
    }
```

```
    //test for new scanlines to the left
```

```
    y1 = y;
```

```
    while(y1 < height && getColor(x, y1) == newColor)
```

```
    {
```

```
        if(x > 0 && getColor(x - 1, y1) == oldColor)
```

```
        {
```

```
            floodFillScanLine(x - 1, y1, newColor, oldColor);
```

```
        }
```

```
        y1++;
```

```
    }
```

```
    y1 = y - 1;
```

```
    while(y1 >= 0 && getColor(x, y1) == newColor)
```

```
    {
```

```
        if(x > 0 && getColor(x - 1, y1) == oldColor)
```

```
        {
```

```
            floodFillScanLine(x - 1, y1, newColor, oldColor);
```

```
        }
```

```
        y1--;
```

```
    }
```

```
    //test for new scanlines to the right
```

```
    y1 = y;
```

```
    while(y1 < height && getColor(x, y1) == newColor)
```

```
    {
```

```

    if(x < width - 1 && getColor(x + 1, y1) == oldColor)
    {
        floodFillScanLine(x + 1, y1, newColor, oldColor);
    }
    y1++;
}
y1 = y - 1;
while(y1 >= 0 && getColor(x, y1) == newColor)
{
    if(x < width - 1 && getColor(x + 1, y1) == oldColor)
    {
        floodFillScanLine(x + 1, y1, newColor, oldColor);
    }
    y1--;
}
}

```

对于上述的递归程序，可以改写为非递归基于栈的扫描线泛洪填，改写以后的代码如下：

```

public void floodFillScanLineWithStack(int x, int y, int newColor, int oldColor)
{
    if(oldColor == newColor) {
        System.out.println("do nothing !!!, filled area!!");
        return;
    }
    emptyStack();

    int y1;
    boolean spanLeft, spanRight;
    push(x, y);

    while(true)
    {
        x = popx();
        if(x == -1) return;
        y = popy();
        y1 = y;
        while(y1 >= 0 && getColor(x, y1) == oldColor) y1--; // go to line top/bottom
        y1++; // start from line starting point pixel
        spanLeft = spanRight = false;
        while(y1 < height && getColor(x, y1) == oldColor)
        {
            setColor(x, y1, newColor);
            if(!spanLeft && x > 0 && getColor(x - 1, y1) == oldColor) // just keep
                left line once in the stack
            {
                push(x - 1, y1);
                spanLeft = true;
            }
            else if(spanLeft && x > 0 && getColor(x - 1, y1) != oldColor)
            {

```



```

        spanLeft = false;
    }
    if(!spanRight && x < width - 1 && getColor(x + 1, y1) == oldColor) //
        just keep right line once in the stack
    {
        push(x + 1, y1);
        spanRight = true;
    }
    else if(spanRight && x < width - 1 && getColor(x + 1, y1) != oldColor)
    {
        spanRight = false;
    }
    y1++;
}
}
}

```

其中 push()、popx()、popy() 为栈进出的三个相关操作, 完整的泛洪填充算法 FloodFillAlgorithm.java 与测试 UI 程序 FloodFillUI.java 的源代码参见源文件的 10-3。

10.4 连通组件标记算法

连通组件标记算法是图像分析最常用的算法之一, 通常被用来寻找与标记连通的区域。连通组件标记本质上是给图像中每个像素指定一个标记值, 同一个连通区域所有像素的标记值保持一致, 直到图像中所有的像素都被标记为止, 通过合并标记使每个连通区域最终只拥有唯一标识。图 10-3 左边为二值图像四个连通区域, 右边为组件标记以后的结果。

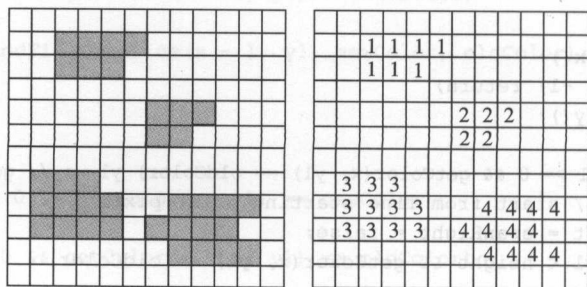


图 10-3 连通区域与标记

从图 10-3 中还可以看出, 不同的连通区域里的像素被标记为不同的值, 相同的区域有相同的标记值。连通组件标记需要遍历连通区域的每个像素。图的遍历实现分为广度优先与深度优先, 本节将利用图的遍历算法实现二值图像的连通组件标记算法。

1. 图的广度优先搜索与深度优先搜索

图的广度优先搜索算法是从一个节点开始, 访问所有相邻节点, 然后从这些相邻节点再

出发,访问所有相邻的子节点,如此迭代直到图中所有的节点都被访问为止。下面是一个广度优先遍历无向图所有节点的例子(如图 10-4 所示)。

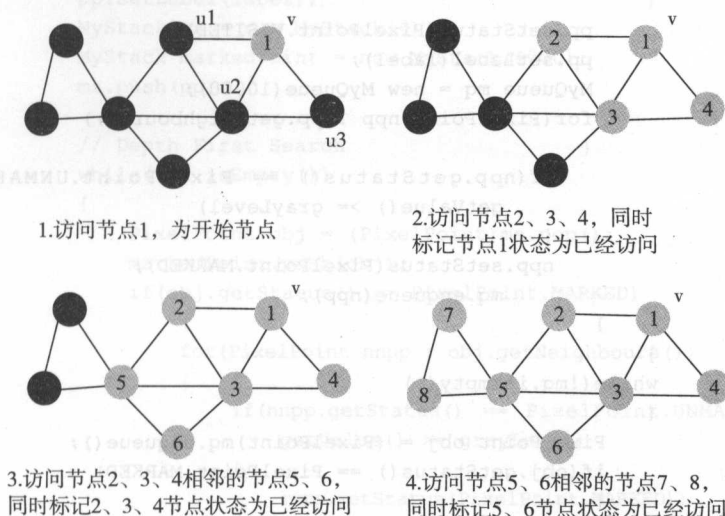


图 10-4 节点遍历过程

图的深度优先搜索则是从一个点出发寻找与该点相连的节点,标记之后,继续从相连节点出发直到找不到任何相连的而且未标记的节点为止。假设无向图如图 10-5 所示。

从节点 1 开始出发,通过深度优先搜索,先到节点 4,然后到节点 2,再到节点 5,最后到节点 3,到节点 3 后再也找不到任何相连而且未标记的节点了,则节点 3 状态设置为已经访问,然后返回节点 5,同样将其设置为已经访问,以此类推直到节点 1 设置为已经访问。从上述流程

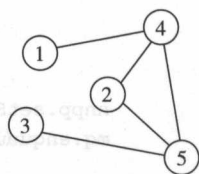


图 10-5 节点示意图

中不难看出,广度优先搜索算法是基于队列的先进先出,而深度优先搜索则是基于栈结构的先进后出。基于 Java 语言实现简单的栈与队列代码可以参见源文件中 10-4 里的 MyStack.java 与 MyQueue.java,这里不再赘述。如果把图像中的每个像素看作图的每个节点,则可以运用图的广度或深度优先搜索实现连通组件的寻找。所以对每个像素可以设置初始节点状态与标记值,将像素点表示为图节点的 Java 语言代码描述同样可参见源文件中 10-4 里的 PixelPoint.java。获取二值图像所有像素初始化以后,得到像素节点数组,基于队列实现广度优先搜索算法的代码如下:

```
public void process()
{
    if(this.pixellist == null) return;
    int label = 1;
    for(PixelPoint pp : pixellist)
    {
```

```

if(pp.getValue() >= grayLevel)
{
    if(pp.getStatus() == PixelPoint.UNMARKED)
    {
        pp.setStatus(PixelPoint.VISITED);
        pp.setLabel(label);
        MyQueue mq = new MyQueue(10000);
        for(PixelPoint npp : pp.getNeighbours())
        {
            if(npp.getStatus() == PixelPoint.UNMARKED && npp.
                getValue() >= grayLevel)
            {
                npp.setStatus(PixelPoint.MARKED);
                mq.enqueue(npp);
            }
        }
        while(!mq.isEmpty())
        {
            PixelPoint obj = (PixelPoint)mq.dequeue();
            if(obj.getStatus() == PixelPoint.MARKED)
            {
                obj.setLabel(label);
                obj.setStatus(PixelPoint.VISITED);
            }
            for(PixelPoint nnpp : obj.getNeighbours())
            {
                if(nnpp.getStatus() == PixelPoint.UNMARKED && nnpp.
                    getValue() >= grayLevel)
                {
                    nnpp.setStatus(PixelPoint.MARKED);
                    mq.enqueue(nnpp);
                }
            }
            label++;
        }
    }
}
}
}
}
}

```

基于栈实现深度优先搜索的代码如下：

```

public void process()
{
    if(this.pixelList == null) return;
    int label = 1;
    for(PixelPoint pp : pixelList)
    {
        if(pp.getValue() >= grayLevel)
        {
            if(pp.getStatus() == PixelPoint.UNMARKED)

```

```

{
    // initialization stack
    pp.setStatus(PixelPoint.MARKED);
    pp.setLabel(label);
    MyStack ms = new MyStack(4);
    MyStack markedPoint = new MyStack(4000);
    ms.push(pp);

    // Depth First Search
    while(!ms.isEmpty())
    {
        PixelPoint obj = (PixelPoint)ms.pop();
        markedPoint.push(obj);
        if(obj.getStatus() == PixelPoint.MARKED)
        {
            for(PixelPoint nnpp : obj.getNeighbours())
            {
                if(nnpp.getStatus() == PixelPoint.UNMARKED && nnpp.
                    getValue() >= grayLevel)
                {
                    nnpp.setStatus(PixelPoint.MARKED);
                    ms.push(nnpp);
                }
            }
        }
    }

    // tag label now!!
    while(!markedPoint.isEmpty())
    {
        PixelPoint obj = (PixelPoint)markedPoint.pop();
        obj.setLabel(label);
        obj.setStatus(PixelPoint.VISITED);
    }
    label++;
}
}
}

```

像素节点的状态有三种，分别为 UNMARKED，表示节点未标记；MARKED，表示节点已经标记但是未完成访问；VISITED，表示节点已经被访问过。完整的广度优先搜索与深度优先搜索算法的代码实现参见源文件中 10-4 里的 BFSAlgorithm.java 与 DFSAlgorithm.java。

2. 基于广度或深度优先搜索实现连通组件标记

基于图的广度或深度优先搜索算法实现连通组件标记算法的代码如下。

1) 实现像素节点初始化及上下左右这四个相连像素节点的添加，代码如下：

```
// 初始化每个像素节点状态
```

```

for(int row=0; row<height; row++) {
    for(int col=0; col<width; col++) {
        index = row * width + col;
        PixelPoint p = new PixelPoint(row, col, (inPixels[index] >> 16) & 0xff);
        pixellist.add(p);
    }
}
// 添加每个像素节点的四邻域像素
for(int row=0; row<height; row++) {
    for(int col=0; col<width; col++) {
        index = row * width + col;
        PixelPoint p = pixellist.get(index);

        // add four neighbors for each pixel
        if((row - 1) >= 0)
        {
            index = (row-1) * width + col;
            p.addNeighbour(pixellist.get(index));
        }
        if((row + 1) < height)
        {
            index = (row+1) * width + col;
            p.addNeighbour(pixellist.get(index));
        }
        if((col - 1) >= 0)
        {
            index = row * width + col-1;
            p.addNeighbour(pixellist.get(index));
        }
        if((col+1) < width)
        {
            index = row * width + col+1;
            p.addNeighbour(pixellist.get(index));
        }
    }
}

```

对所有像素节点完成初始化以后就可以进行下一步处理了。

2) 通过广度或深度优先搜索实现连通组件标记, 以第一步的结果为输入参数, 实现连通组件标记与查找, 深度优先搜索实现代码如下:

```

// 深度优先搜索算法, 连通组件标记
DFSAlgorithm dfs = new DFSAlgorithm(pixellist);
dfs.process();
System.out.println("Total Number of Labels : " + dfs.getTotalOfLabels());

```

广度优先搜索实现代码如下:

```

BFSAlgorithm bfs = new BFSAlgorithm(pixellist);
bfs.process();

```

```
System.out.println("Total Number of Labels : " + bfs.getTotalOfLabels());
```

3) 连通组件随机着色与显示, 完成第二步以后得到的连通组件被标记为不同的值, 对这些不同的连通区域需要显示不同颜色, 这里通过随机生成每个连通区域的颜色实现连通组件着色与显示, 实现代码如下:

```
// post process——区域连通组件着色
for(int row=0; row<height; row++) {
    int ta = 0, tr = 0, tg = 0, tb = 0;
    for(int col=0; col<width; col++) {
        index = row * width + col;
        PixelPoint p = pixelList.get(index);
        ta = 255;
        if(p.getLabel() > 0)
        {
            Color c = getColor(p.getLabel());
            tr = c.getRed();
            tg = c.getGreen();
            tb = c.getBlue();
        }
        else
        {
            tr = p.getValue();
            tg = p.getValue();
            tb = p.getValue();
        }
        outPixels[index] = (ta << 24) | (tr << 16) | (tg << 8) | tb;
    }
}
```

根据标记值的不同, 生成标记颜色方法的代码如下:

```
private Color getColor(Integer label)
{
    Color c = colorMap.get(label);
    if(c == null)
    {
        int red = (int)(Math.random() * 255);
        int green = (int)(Math.random() * 255);
        int blue = (int)(Math.random() * 255);
        c = new Color(red, green, blue);
        colorMap.put(label, c);
    }
    return c;
}
```

完整连通组件标记算法的代码参见源文件中 10-4 里的 LabelledConnectedRegionAlg.java。实现图像连通组件标记的算法还可以基于递归扫描先标记后检查合并标记的方法, 感兴趣的读者可以自己实现。

10.5 二值图像边缘跟踪

二值图像作为一种特殊的图像形态，其边缘提取相对简单，如果有噪声，首先可以基于组件标记算法去掉像素总数小于阈值 T 的噪声区域，或者通过高斯模糊来降低噪声影响，然后就可以对图像进行边缘提取，二值图像的边缘提取可以采用拉普拉斯算子一次完成，关于拉普拉斯边缘提取的知识可以参见第 9 章的相关内容，这里不再赘述。有图像边缘以后，通过对边缘跟踪处理实现图像对象轮廓提取，为进一步识别匹配做准备。实现对简单对象图像轮廓提取的边缘跟踪算法常见的有两种，分别是 Square Tracing 与 Moore-Neighbour 算法，下面就分别详细介绍。

1. Square Tracing 算法

Square Tracing 边缘跟踪的主要思想是假设图像背景为白色，前景像素为黑色，从底向上，从左到右地扫描每个像素，扫描到第一个黑色像素标记为边缘开始像素点，然后执行如下操作：

每次扫描到一个黑色像素时，继续向左扫描；每次扫描到白色背景像素时，继续向右扫描直到再次遇到被标记的起始黑色像素，结束扫描。

完整实现基于边缘跟踪的轮廓提取程序如下。

1) 初始化，获取二值图像像素数组，其实现代码如下：

```
//初始化，获取输入图像像素数组
int[] inPixels = new int[width * height];
int[] outPixels = new int[width * height];
getRGB(src, 0, 0, width, height, inPixels);
```

2) 自下而上，由左到右扫描像素数组，找到起始像素点 StartPixel，代码如下：

```
MyQueue mq = new MyQueue(100000);
PixelPoint startP = null;
boolean foundStartP = false;
// 从左到右
for (int col = 0; col < width; col++) {
    // 至底向上
    for (int row = height-1; row >= 0; row--) {
        // 获取像素值
        int g1 = getPixel(inPixels, width, height, col, row);
        if(g1 == 0 && startP == null)
        {
            startP = new PixelPoint(row, col, g1);
            mq.enqueue(startP);
            foundStartP = true;
            break;
        }
    }
    if(foundStartP)
    {
```

```
break;
```

```
}
```

```
}
```

3) 从起始像素点开始, 向左或向右扫描, 得到边缘像素则添加到边缘队列中, 直到又重新回到开始像素 StartPixel, 代码如下:

```
PixelPoint currentP = null;
while(!samePixel(currentP, startP))
```

```
{
```

```
    if(currentP == null)
```

```
    {
```

```
        // 初始化开始像素
```

```
        int xp = startP.getX();
```

```
        int yp = startP.getY();
```

```
        // System.out.println("xp : " + xp + " yp : " + yp);
```

```
        int lxp = xp - 1;
```

```
        int gr = getPixel(inPixels, width, height, lxp, yp);
```

```
        currentP = new PixelPoint(yp, lxp, gr);
```

```
        currentP.setLabel(LEFT_X);
```

```
    }
```

```
    else
```

```
    {
```

```
        int direction = currentP.getLabel();
```

```
        int xp = currentP.getX();
```

```
        int yp = currentP.getY();
```

```
        // System.out.println("xp : " + xp + " yp : " + yp);
```

```
        // 发现边缘像素
```

```
        if(currentP.getValue() == 0)
```

```
        {
```

```
            // turn left
```

```
            mq.enqueue(currentP);
```

```
            if(direction == LEFT_X)
```

```
            {
```

```
                yp = yp + 1;
```

```
                direction = DOWN_Y;
```

```
            }
```

```
            else if(direction == RIGHT_X)
```

```
            {
```

```
                yp = yp - 1;
```

```
                direction = UPPER_Y;
```

```
            }
```

```
            else if(direction == UPPER_Y)
```

```
            {
```

```
                xp = xp - 1;
```

```
                direction = LEFT_X;
```

```
            }
```

```
            else
```

```
            {
```

```
                xp = xp + 1;
```

```
                direction = RIGHT_X;
```

```
            }
```

```
        }
```

```

10.5 二值图像边缘跟踪
else
{
    // 非边缘像素，继续寻找
    if(direction == LEFT_X)
    {
        yp = yp - 1;
        direction = UPPER_Y;
    }
    else if(direction == RIGHT_X)
    {
        yp = yp + 1;
        direction = DOWN_Y;
    }
    else if(direction == UPPER_Y)
    {
        xp = xp + 1;
        direction = RIGHT_X;
    }
    else
    {
        xp = xp - 1;
        direction = LEFT_X;
    }
}
// 设定当前像素值
int gr = getPixel(inPixels, width, height, xp, yp);
currentP = new PixelPoint(yp, xp, gr);
currentP.setLabel(direction);
}
}

```

4) 初始化背景像素为白色，显示边缘像素队列中每个像素，得到输出结果：

```

// 白色背景
Arrays.fill(outPixels, -1);
while(!mq.isEmpty())
{
    PixelPoint edgePixel = (PixelPoint)mq.dequeue();
    int col = edgePixel.getX();
    int row = edgePixel.getY();
    setPixel(outPixels, width, height, col, row, 0);
}

```

完整的 Square Tracing 算法源代码 SquareTraceAlgorithm.java 参见本书源文件中的 10-5。其运行结果如图 10-6 所示（左边为原图，右边为对象轮廓）。

值得注意的是，Square Tracing 算法本质上是一种基于四邻域链接寻找边缘跟踪的算法，而且其停止条件过于简单，一个提高方法就是改变停止条件，即

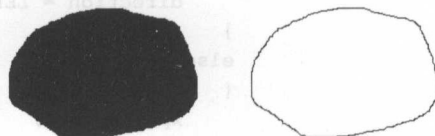


图 10-6 轮廓提取

假设经过开始像素两次才停止，或者两次进入开始像素的方向完全一致才停止继续边缘寻找。感兴趣的读者可以自己进一步尝试。

2. Moore-Neighbor 跟踪算法

该算法基于 Jacob 停止条件实现边缘轮廓提取，是一种非常理想的对象轮廓提取算法，在具体介绍该算法之前，首先要明确什么才是一个像素的 Moore-Neighbor 像素，一个像素的周围八个邻域像素分别被命名为 $P1 \sim P8$ ，如图 10-7 所示。

	$P1$	$P2$	$P3$	
	$P8$	P	$P4$	
	$P7$	$P6$	$P5$	

图 10-7 邻域像素

其中 P 为当前像素。Moore-Neighbor 跟踪算法的思想是假设二值图像为白色背景，连通组件为黑色像素，现在自底向上、从左到右扫描每一列的每个像素，当扫描到黑色像素 P 时，进行标记，然后回退到上一个扫描像素位置，沿着顺时针方向扫描 P 周围的像素，如果遇到下一个黑色像素则继续重复该动作，直到第二次经过起始扫描到的黑色像素为止。所有被标记的黑色像素组成的像素点即为对象轮廓。程序实现基于 Moore-Neighbor 边缘跟踪对象轮廓提取算法步骤如下：

1) 初始化二值图像获取图像像素数组，代码如下。

```
//初始化，获取输入图像像素数组
int[] inPixels = new int[width * height];
int[] outPixels = new int[width * height];
getRGB(src, 0, 0, width, height, inPixels);
```

2) 自下而上、从左到右，逐列扫描每个像素，在遇到第一个黑色像素时，标记其为起始像素，代码如下：

```
// 从左到右
for (int col = 0; col < width; col++) {
    // 至底向上
    for (int row = height-1; row >= 0; row--) {
        // 获取像素值
        int g1 = getPixel(inPixels, width, height, col, row);
        if (g1 == 0 && startP == null)
        {
            startP = new PixelPoint(row, col, g1);
            mq.enqueue(startP);
            foundStartP = true;
            break;
        }
    }
    if (foundStartP)
    {
        break;
    }
}
```

3) 从起始像素开始按原方向回退到前一个像素，然后顺时针扫描周围的八个像素，如果

遇到黑色像素, 标记之后重复上述动作, 直到遇到起始标记像素则结束, 代码如下。

```

PixelPoint currentP = null;
while(!samePixel(currentP, startP))
{
    if(currentP == null)
    {
        // 初始化开始像素, 回退到上一个像素
        int xp = startP.getX();
        int yp = startP.getY();
        yp = yp + 1;
        int gr = getPixel(inPixels, width, height, xp, yp);
        currentP = new PixelPoint(yp, xp, gr);
        currentP.setLabel(P6);
    }
    else
    {
        int position = currentP.getLabel();
        int xp = currentP.getX();
        int yp = currentP.getY();
        // 发现边缘像素
        if(currentP.getValue() == 0)
        {
            // 回退
            mq.enqueue(currentP);
            if(position == P1)
            {
                yp = yp + 1;
                position = P6;
            }
            else if(position == P2)
            {
                xp = xp - 1;
                position = P8;
            }
            else if(position == P3)
            {
                xp = xp - 1;
                position = P8;
            }
            else if(position == P4)
            {
                yp = yp - 1;
                position = P2;
            }
            else if(position == P5)
            {
                yp = yp - 1;
                position = P2;
            }
            else if(position == P6)
            {

```

```

1) 至少 P2、P4      xp = xp + 1;  // 白色像素
2) 至少 P4、P6      position = P4;
3) 至少 P4、P6      }
4) 至少 P2、P4      else if(position == P7)
5) 至少 P2、P4      {
6) 至少 P2、P4      xp = xp + 1;
7) 至少 P2、P4      position = P4;
8) 至少 P2、P4      }
9) 至少 P2、P4      else // P8
10) 至少 P2、P4     {
11) 至少 P2、P4     xp = xp + 1;
12) 至少 P2、P4     position = P6;
13) 至少 P2、P4     }
14) 至少 P2、P4     }
15) 至少 P2、P4     else
16) 至少 P2、P4     { // 非边缘像素, 顺时针方向,
17) 至少 P2、P4     if(position == P1)
18) 至少 P2、P4     {
19) 至少 P2、P4     xp = xp + 1;
20) 至少 P2、P4     position = P2;
21) 至少 P2、P4     }
22) 至少 P2、P4     else if(position == P2)
23) 至少 P2、P4     {
24) 至少 P2、P4     xp = xp + 1;
25) 至少 P2、P4     position = P3;
26) 至少 P2、P4     }
27) 至少 P2、P4     else if(position == P3)
28) 至少 P2、P4     {
29) 至少 P2、P4     yp = yp + 1;
30) 至少 P2、P4     position = P4;
31) 至少 P2、P4     }
32) 至少 P2、P4     else if(position == P4)
33) 至少 P2、P4     {
34) 至少 P2、P4     yp = yp + 1;
35) 至少 P2、P4     position = P5;
36) 至少 P2、P4     }
37) 至少 P2、P4     else if(position == P5)
38) 至少 P2、P4     {
39) 至少 P2、P4     xp = xp - 1;
40) 至少 P2、P4     position = P6;
41) 至少 P2、P4     }
42) 至少 P2、P4     else if(position == P6)
43) 至少 P2、P4     {
44) 至少 P2、P4     xp = xp - 1;
45) 至少 P2、P4     position = P7;
46) 至少 P2、P4     }
47) 至少 P2、P4     else if(position == P7)
48) 至少 P2、P4     {
49) 至少 P2、P4     yp = yp - 1;
50) 至少 P2、P4     position = P8;
51) 至少 P2、P4     }
52) 至少 P2、P4     }
53) 至少 P2、P4     }
54) 至少 P2、P4     }
55) 至少 P2、P4     }
56) 至少 P2、P4     }
57) 至少 P2、P4     }
58) 至少 P2、P4     }
59) 至少 P2、P4     }
60) 至少 P2、P4     }
61) 至少 P2、P4     }
62) 至少 P2、P4     }
63) 至少 P2、P4     }
64) 至少 P2、P4     }
65) 至少 P2、P4     }
66) 至少 P2、P4     }
67) 至少 P2、P4     }
68) 至少 P2、P4     }
69) 至少 P2、P4     }
70) 至少 P2、P4     }
71) 至少 P2、P4     }
72) 至少 P2、P4     }
73) 至少 P2、P4     }
74) 至少 P2、P4     }
75) 至少 P2、P4     }
76) 至少 P2、P4     }
77) 至少 P2、P4     }
78) 至少 P2、P4     }
79) 至少 P2、P4     }
80) 至少 P2、P4     }
81) 至少 P2、P4     }
82) 至少 P2、P4     }
83) 至少 P2、P4     }
84) 至少 P2、P4     }
85) 至少 P2、P4     }
86) 至少 P2、P4     }
87) 至少 P2、P4     }
88) 至少 P2、P4     }
89) 至少 P2、P4     }
90) 至少 P2、P4     }
91) 至少 P2、P4     }
92) 至少 P2、P4     }
93) 至少 P2、P4     }
94) 至少 P2、P4     }
95) 至少 P2、P4     }
96) 至少 P2、P4     }
97) 至少 P2、P4     }
98) 至少 P2、P4     }
99) 至少 P2、P4     }
100) 至少 P2、P4     }

```



```

else
{
    yp = yp - 1;
    position = P1;
}
}
// 设定当前像素值
int gr = getPixel(inPixels, width, height, xp, yp);
currentP = new PixelPoint(yp, xp, gr);
currentP.setLabel(position);
}
}

```

4) 对标记的所有黑色像素着色显示, 得到对象轮廓。

```

// 白色背景
Arrays.fill(outPixels, -1);
while(!mq.isEmpty())
{
    PixelPoint edgePixel = (PixelPoint)mq.dequeue();
    int col = edgePixel.getX();
    int row = edgePixel.getY();
    setPixel(outPixels, width, height, col, row, 0);
}

```

完整的基于 Moore-Neighbour 边缘跟踪算法的轮廓提取源代码 MooreNeighbourTrace-Algorithm.java 参见源文件中的 10-5。关于中止条件, 可以改为两次经过起始像素点, 这样大部分的对象都可以实现边缘跟踪。

本节主要介绍了两种简单的对象边缘跟踪算法, 基于边缘跟踪可以实现连通组件的轮廓提取, 这些都在现实应用中具有实际意义。

10.6 二值图像细化

二值图像细化是图像骨架提取的常用手段之一, 可通过细化得到图像对象骨架, 为下一步处理做好准备。最常见的基于模板扫描方式的二值图像细化算法是 Zhang-Suen thinning 算法, 它具有运算速度快、实现简单的优点。Zhang-Suen thinning 算法假设黑色像素为 1、白色背景像素为 0, 输入像素数组为 $N \times M$ 大小, 假设所有黑色像素 P 有如图 10-8 所示的八邻域像素。

定义 P 的连接数 $Connectivity(P)$ 为 $P_2 \rightarrow P_9 \rightarrow P_2$ 顺序排列中所有 $0 \rightarrow 1$ 出现次数、定义 P 周围黑色像素个数 $Black(P)$ 为 $P_2 \sim P_9$ 中黑色像素的数目。所有黑色像素 P 满足下面第一个子迭代的所有条件:

- 1) 连接数目 $Connectivity(P) = 1$ 。
- 2) 邻域中黑色像素数目 $2 \leq Black(P) \leq 6$ 。

P1	P2	P3	
P8	P	P4	
P7	P6	P5	

图 10-8 八个像素顺序

3) 至少 P_2 、 P_4 、 P_6 之中有一个是白色像素。

4) 至少 P_4 、 P_6 、 P_8 之中有一个是白色像素。

设置该像素为白色像素。继续对剩下的黑色像素进行分析, 如果满足第二个子迭代的所有条件:

5) 连接数目 $Connectivity(P) = 1$ 。

6) 邻域中黑色像素数目 $2 \leq Black(P) \leq 6$ 。

7) 至少 P_2 、 P_4 、 P_8 之中有一个是白色像素。

8) 至少 P_2 、 P_6 、 P_8 之中有一个是白色像素。

设置该像素为白色像素, 直到所有的黑色像素经过上述两个子迭代过程再无改变, 则结束迭代, 细化完成。编程实现该算法大致可以分为如下几步:

1) 获取输入二值图像像素数组 $N \times M$ 大小 (假设白色为背景像素, 黑色为对象组件)。

2) 初始化像素, 计算每个黑色像素的连接数目与邻域中的黑色像素数目。

3) 循环对每个黑色像素完成第一子迭代条件检测, 符合条件则设像素值为白色。

4) 重复步骤 2。

5) 循环对每个黑色像素完成第二子迭代条件检测, 符合条件则设像素为白色。

6) 循环步骤 2~5 直到黑色像素值不再改变为止。

7) 输出处理以后的数组。

完整实现上述步骤的代码片段如下:

```
// 获取输入像素数组
pixelList = new ArrayList<ThinPixel>();
int[] inPixels = new int[width*height];
int[] outPixels = new int[width*height];
getRGB( src, 0, 0, width, height, inPixels );
boolean changed = true;
// 开始细化
while(changed)
{
    changed = false;
    // 初始每个黑色像素
    initPixels(inPixels, width, height);
    // 迭代一: 条件
    for(ThinPixel tp : pixelList)
    {
        if(tp.getValue() == 0) continue;
        int p246 = tp.getP2() * tp.getP4() * tp.getP6();
        int p468 = tp.getP8() * tp.getP4() * tp.getP6();
        if((tp.getNumOfBlack() >= 2 && tp.getNumOfBlack() <= 6) &&
            (p468 == 0) && (p246 == 0) &&
            (tp.getNumOfConnectivity() == 1)) {
            setPixel(inPixels, width, height, tp.getCol(), tp.getRow(), 255);
            changed = true;
        }
    }
}
```

```

    }
    // 初始化
    initPixels(inPixels, width, height);
    // 迭代二：条件检测
    for(ThinPixel tp : pixelList)
    {
        if(tp.getValue() == 0) continue;
        int p248 = tp.getP2() * tp.getP4() * tp.getP8();
        int p268 = tp.getP2() * tp.getP6() * tp.getP8();
        if((tp.getNumOfBlack() >= 2 && tp.getNumOfBlack() <= 6) &&
            (p248 == 0) &&
            (p268 == 0) &&
            (tp.getNumOfConnectivity() == 1)) {
            changed = true;
            setPixel(inPixels, width, height, tp.getCol(), tp.getRow(), 255);
        }
    }
}
// 输出结果数组
Arrays.fill(outPixels, -1);
for(ThinPixel tp : pixelList)
{
    int row = tp.getRow();
    int col = tp.getCol();
    int p = tp.getValue();
    if(p==0)
    {
        p = 255;
    }
    else
    {
        p = 0;
    }
    setPixel(outPixels, width, height, col, row, p);
}

```

其中 `initPixels` 方法是实现每个黑色像素的初始化处理，该方法的代码如下：

```

private void initPixels(int[] inPixels, int width, int height)
{
    int index = 0;
    pixelList.clear();
    for(int row=0; row<height; row++) {
        for(int col=0; col<width; col++) {
            index = row * width + col;
            int value = (inPixels[index] >> 16) & 0xff;
            if(value == 255)
            {
                ThinPixel p = new ThinPixel(row, col, 0); // white;
                pixelList.add(p);
            }
        }
    }
}

```

```

    }
    else
    {
        ThinPixel p = new ThinPixel(row, col, 1); // black;
        pixelList.add(p);
    }
}

for(ThinPixel tp : pixelList)
{
    int row = tp.getRow();
    int col = tp.getCol();

    if(tp.getValue() == 0) continue;
    // 寻找细化目标像素点
    int p2 = getPixel(inPixels, width, height, col, row-1);
    int p3 = getPixel(inPixels, width, height, col+1, row-1);
    int p4 = getPixel(inPixels, width, height, col+1, row);
    int p5 = getPixel(inPixels, width, height, col+1, row+1);
    int p6 = getPixel(inPixels, width, height, col, row+1);
    int p7 = getPixel(inPixels, width, height, col-1, row+1);
    int p8 = getPixel(inPixels, width, height, col-1, row);
    int p9 = getPixel(inPixels, width, height, col-1, row-1);

    p2 = p2 == 0 ? 1 : 0;
    p3 = p3 == 0 ? 1 : 0;
    p4 = p4 == 0 ? 1 : 0;
    p5 = p5 == 0 ? 1 : 0;
    p6 = p6 == 0 ? 1 : 0;
    p7 = p7 == 0 ? 1 : 0;
    p8 = p8 == 0 ? 1 : 0;
    p9 = p9 == 0 ? 1 : 0;

    int sum = p2+p3+p4+p5+p6+p7+p8+p9;
    int times = 0;
    if(p2==0 && p3 == 1)
    {
        times++;
    }
    if(p3==0 && p4== 1)
    {
        times++;
    }
    if(p4==0 && p5 == 1)
    {
        times++;
    }
    if(p5== 0 && p6==1)
    {
        times++;
    }
}

```

```

if(p6== 0 && p7==1)
{
    times++;
}
if(p7== 0 && p8==1)
{
    times++;
}
if(p8== 0 && p9==1)
{
    times++;
}
if(p9== 0 && p2==1)
{
    times++;
}
tp.setNumOfConnectivity(times);
tp.setNumOfBlack(sum);
tp.setP2(p2);
tp.setP4(p4);
tp.setP6(p6);
tp.setP8(p8);
}
}

```

完整的 Zhang-Suen thinning 算法源代码参加源文件中 10-6 的 ZhangSuenThinningAlg.java。实际运行效果如图 10-9 所示。

图 10-9 中，左边为输入文字，右边为细化以后的文字。

图像处理 图像处理

图 10-9 细化结果

10.7 计算连通区域几何质心

在具体介绍计算方法之前，首先简单介绍一下物体的几何质心概念。质心就是通过该点，使区域达到一种质量上的平衡状态。对于质心的概念，可能物理学上讲得比较多，简单地讲就是规则几何物体的中心，不规则的可以通过挂绳子的方法来寻找，如图 10-10 所示。

1. 基本原理

假设二值图像像素数组大小为 $N \times M$ ，二值图像连通区域几何质心计算使用如下数学公式：

$$\text{计算区域总像素 } A = \sum_{i=1}^N \sum_{j=1}^M B[i, j]$$

这里，假设 $B[i, j]$ 属于该连通区域，则 $B[i, j] = 1$ ，否则为 0。

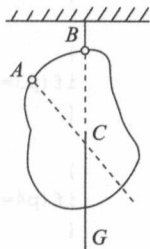


图 10-10 几何质心

根据上述计算结果质心坐标为: $X_0 = \frac{\sum_{i=1}^N \sum_{j=1}^M jB[i,j]}{A}$ 、 $Y_0 = \frac{\sum_{i=1}^N \sum_{j=1}^M iB[i,j]}{A}$

这里依据的数学模型是图像 Moments, Moments 本质上基于权重的像素计算, 它对图像中的连通区域或分割对象的各种属性计算非常实用。常见的图像 Moments 的表达式如下:

$$M_{pq} = \sum_{i=1}^N \sum_{j=1}^M i^p j^q f(i,j)$$

其中, p 与 q 值之和称为 $(p+q)^{th}$ 阶 Moments。计算连通区域几何质心正是运用第 0 阶与 1 阶 Moments 的结果计算得到。本书后面还会对 Moments 做更多的介绍, 这里就不再进一步探讨。

2. 实现步骤

基于 Moments 实现连通区域质心计算的完整代码实现大致可以分为如下几步:

- 1) 获取输入二值图像像素数组。
- 2) 通过连通组件标记算法找到所有的连通区域, 并分别标记。
- 3) 对每个连通区域计算第 0 阶与第 1 阶 Moments, 然后得到质心坐标。
- 4) 用不同颜色绘制连通区域的质心, 输出处理后的图像。

在上述编程步骤中, 关于第一步与第二步, 本章已经有详细介绍, 这里不再赘述。第三步利用 Moments 计算质心的实现代码如下:

```
// third step, calculate center point of each region area (connected component)
int[] input = new int[pixelList.size()];
GeometricMomentsAlg momentsAlg = new GeometricMomentsAlg();
momentsAlg.setBackground(0);
double[][] labelCenterPos = new double[max][2];
for(int i=1; i<=max; i++)
{
    for(int p=0; p<input.length; p++)
    {
        if(pixelList.get(p).getLabel() == i)
        {
            input[p] = pixelList.get(p).getLabel();
        }
        else
        {
            input[p] = 0;
        }
    }
    // 计算每个组件的质心
    labelCenterPos[i-1] = momentsAlg.getGeometricCenterCoordinate(input, width, height);
}
```

其中 GeometricMomentsAlg 类是专门用来计算 Moments 阶实现类的, 实现任意 Moments

阶计算的方法代码如下:

```
public double moments(int[] pixels, int width, int height, int p, int q)
{
    double mpq = 0.0;
    int index = 0;
    for(int row=0; row<height; row++)
    {
        for(int col=0; col<width; col++)
        {
            index = row * width + col;
            if(pixels[index] == BACKGROUND) continue;
            mpq += Math.pow(row, p) * Math.pow(col, q);
        }
    }
    return mpq;
}
```

最后一步实现连通区域与质心颜色着色的代码如下:

```
// render the each connected component center position
for(int row=0; row<height; row++) {
    for(int col=0; col<width; col++) {
        index = row * width + col;
        if(pixelList.get(index).getLabel() == 0)
        {
            // make it as black for background
            outPixels[index] = (255 << 24) | (0 << 16) | (0 << 8) | 0;
        }
        else
        {
            // make it as blue for each region area
            outPixels[index] = (255 << 24) | (0 << 16) | (0 << 8) | 100;
        }
    }
}

// make it as white color for each center position
for(int i=0; i<max; i++)
{
    int crow = (int)labelCenterPos[i][0];
    int ccol = (int)labelCenterPos[i][1];
    index = crow * width + ccol;
    outPixels[index] = (255 << 24) | (255 << 16) | (255 << 8) | 255;
}
```

最终输出的图像, 每个区域为蓝色着色, 质心像素为白色表示。完整计算连通区域质心的实现类源代码清单参见源文件中 10-7 的 AreaCenterFilter.java。关于 Moments 计算类的源代码同样参见 10-7 的 GeometricMomentsAlg.java。

10.8 计算连通区域方向角度

Moments 的一阶与零阶用来计算图像连通区域的质心, 二阶可以实现对图像连通区域方向角度的计算, 最终得到的是一个与中心质点和水平线相交的夹角, 即连通区域的方向角度。

1. 基本原理

计算连通区域方向角度时, 同样是基于图像 Moments 得到的计算结果, 与计算图像质心不同, 角度是基于二阶 ($p + q = 2$) 图像 Moments 的结果得到的, 二阶 Moments 的结果除了可用于计算角度 θ 之外, 还可以计算连通区域的扁平程度 E 。同样, 假设二值图像的像素数组为 $N \times M$ 大小, 其计算扁平程度 E 表示为公式:

$$E = a \sin^2 \theta - b \sin \theta \cos \theta + c \cos^2 \theta$$

其中 a 、 b 、 c 三个分别为图像二阶 Moments 计算结果:

$$a = \sum_{i=1}^N \sum_{j=1}^M x_{ij}^2 B[i, j]$$

$$b = \sum_{i=1}^N \sum_{j=1}^M x_{ij} y_{ij} B[i, j]$$

$$c = \sum_{i=1}^N \sum_{j=1}^M y_{ij}^2 B[i, j]$$

假设 $a \neq c$, 然后根据反三角函数知识即可得到连通区域的角度 $\theta = \frac{1}{2} \tan^{-1} \frac{2b}{a-c}$ 。同样进一步可以得到

$$E = \frac{1}{2}(a+c) - \frac{1}{2}(a-c) \cos 2\theta - \frac{1}{2} b \sin 2\theta$$

其中:

$$\sin 2\theta = \pm \frac{b}{\sqrt{b^2 + (a-c)^2}}, \quad \cos 2\theta = \pm \frac{a-c}{\sqrt{b^2 + (a-c)^2}}$$

如果对 $\sin 2\theta$ 与 $\cos 2\theta$ 取正值则得到 E 的最小值 E_{\min} , 取负值则得到 E 的最大值 E_{\max} 。最终得

到扁平程度比率 $e = \frac{E_{\max}}{E_{\min}}$, e 值越接近零则连通区域越趋向线条状, e 值越接近 1 则连通区域越趋向圆形。从上面公式可以看出, 最重要的是计算出 a 、 b 、 c 三个系数值, 也就是要对图像每个连通区域进行二阶 Moments 计算, 然后根据 a 、 b 、 c 三个系数值得到角度大小 θ 与扁平程度比率 e 。

2. 实现步骤

实现连通区域角度方向 θ 与扁平程度比率 (roundness ratio) e 计算与结果显示的程序, 大致需要如下几步:

- 1) 对输入的二值图像获取像素数组 $N \times M$ 大小。
- 2) 通过连通组件标记算法获取并标记每个连通区域。

3) 计算每个连通区域的二阶 Moments 得到 a 、 b 、 c 三个系数值。

4) 根据每个连通区域的 a 、 b 、 c 三个值计算角度方向 θ 与 e 。

5) 根据角度方向 θ 着色显示二值图像结果。

在上述步骤中，关于第一步与第二步，本章前面已经有详细介绍与实现，第三步计算图像连通区域二阶 Moments 得到 a 、 b 、 c 三个系数值，计算每个连通区域的角度 θ 与 E_{\max} 、 E_{\min} 的代码片段如下：

```
// 计算三个参数a,b,c, 角度theta, Emin, Emax
int[] input = new int[pixelList.size()];
GeometricMomentsAlg momentsAlg = new GeometricMomentsAlg();
momentsAlg.setBACKGROUND(0);
double[][] labelCenterPos = new double[max][2];
double[][] centerAngles = new double[max][3];
for(int i=1; i<=max; i++)
{
    for(int p=0; p<input.length; p++)
    {
        if(pixelList.get(p).getLabel() == i)
        {
            input[p] = pixelList.get(p).getLabel();
        }
        else
        {
            input[p] = 0;
        }
    }
    // 计算每个组件的质心
    labelCenterPos[i-1] = momentsAlg.getGeometricCenterCoordinate(input, width, height);
    // 计算每个组件的
    double a = momentsAlg.centralMoments(input, width, height, 0, 2);
    double b = momentsAlg.centralMoments(input, width, height, 1, 1);
    double c = momentsAlg.centralMoments(input, width, height, 2, 0);
    double bb = b*b;
    double ac2 = Math.pow((a-c), 2);
    double sum = 2 * Math.sqrt(bb + ac2);
    double angle = Math.atan((2*b)/(a - c))/2.0;
    double emax=(a+c)/2 - (ac2/sum) - (bb/sum);
    double emin=(a+c)/2 + (ac2/sum) + (bb/sum);
    // 角度范围rescale到0~180之间
    centerAngles[i-1][0] = angle + Math.PI/2.0;
    centerAngles[i-1][1] = emax;
    centerAngles[i-1][2] = emin;
}
```

因为反三角函数 $\tan^{-1}\theta$ 的取值范围在 $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$ 之间，所以对得到的角度值要加上 $\frac{\pi}{2}$ 使得

最终角度 θ 的取值范围为 $[0, \pi]$ 之间。根据得到的角度 θ 与 E_{\max} 、 E_{\min} 的值，对输入像素进行适当的着色处理就可以得到最终的结果，对像素点着色处理显示结果的代码片段如下：

```
int labelCount = centerAngles.length;
for(int i=0; i<labelCount; i++)
{
    System.out.println("Region " + i + "'s angle = " + centerAngles[i][0]);
    System.out.println("Region " + i + " e = " + (centerAngles[i][1]/centerAngles[i][2]));
    double sin = Math.sin(centerAngles[i][0]);
    double cos = Math.cos(centerAngles[i][0]);
    System.out.println("sin = " + sin);
    System.out.println("cos = " + cos);
    System.out.println();
    int crow = (int)labelCenterPos[i][0];
    int ccol = (int)labelCenterPos[i][1];
    int radius = (int)centerAngles[i][1];
    // it is trick, display correct angle as you see!!!
    for(int j=0; j<radius; j++)
    {
        int drow = (int)(crow - j * sin);
        int dcol = (int)(ccol + j * cos);
        if(drow >= height || drow <=0 ) continue;
        if(dcol >= width || dcol <=0 ) continue;
        index = drow * width + dcol;
        outPixels[index] = (255 << 24) | (255 << 16) | (255 << 8) | 0;
    }
    int cx = (int)labelCenterPos[i][1];
    // 根据中心点显示水平直线
    for(int px=cx;px<width; px++)
    {
        int cy = (int)labelCenterPos[i][0];
        index = cy * width + px;
        outPixels[index] = (255 << 24) | (255 << 16) | (255 << 8) | 0;
    }
}
```

上述着色处理中，根据角度计算出每个需要着色的像素，只是简单的三角函数知识的运用，而且都是从质心开始出发的，所以很容易计算。完整的计算连通区域方向角度的类源代码参见本书源文件中 10-8 的 AreaOrientationFilter.java。代码已经经过测试，读者也可以运行修改。关于图像 Moments 计算图像角度方向的内容就介绍到这里，感兴趣的读者可以自己进一步阅读相关资料。

10.9 小结

本章介绍了二值图像的基本概念、采样问题，然后由浅入深地介绍了二值图像的泛洪

合。



第 11 章 Chapter 11

图像形态学

上一章介绍了二值图像处理的相关知识，在实际应用中，二值图像有一类重要处理方法，就是基于图像形态学的处理。图像形态学操作是图像像素集合的结构化操作，常见的为二值图像与灰度图像，为了减少不必要的复杂性，本章所讲的各种形态学操作都是以二值图像为对象的。

首先从形态学的基本概念——腐蚀与膨胀开始，进而讲解开闭操作、基于形态学的边缘检测、距离变换与骨架提取、区域填充等知识，这些知识的学习将进一步帮助读者提高对二值图像处理的认知水平。同样本章每个知识点的内容介绍会涉及一些数学上集合的基本概念与知识，提前学习与掌握简单的数学集合操作基本概念与知识有助于读者更好地理解本章内容。

11.1 像素集合操作

图像形态学是指基于像素集合结构化操作实现图像的各种处理，在具体介绍形态学处理之前，首先介绍一下像素集合各种操作的定义与数学表示。如果把图像像素数组看成一个集合，常见的集合操作并、交、补、差、反等都可以基于像素集合完成。假设 A 与 B 是两个集合，且像素点 a 在集合 A 之中，那么为 $a \in A$ ，如果像素点 a 不在集合 A 中则表示为 $a \notin A$ ，如果在 A 中的每个像素点同时也在 B 中，则 A 可以看成 B 的子集，表示为 $A \subseteq B$ 。本节所有关于集合的操作前提是假设像素数组大小为 $N \times M$ ，像素点索引 $p = x + Ny$ ，其中 x 与 y 分别是列与行索引。

1. 集合操作：并

集合并操作可以表示 $C = A \cup B$ ，其中集合 A 与集合 B 是如图 11-1 所示的两张二值图像

的像素数组集合。

其中黑色像素表示为零，白色像素表示为 1 (255)。并操作完成以后，得到的结果如图 11-3 所示。

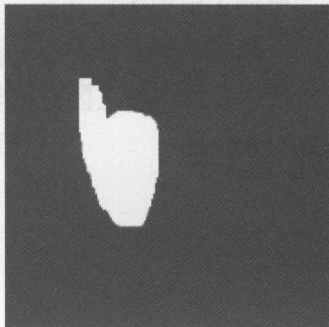


图 11-1 像素集合 A



图 11-2 像素集合 B

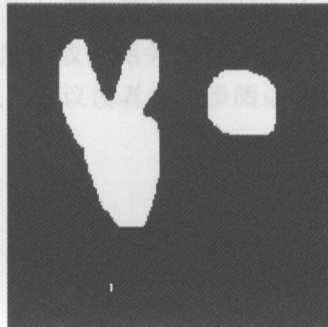


图 11-3 并操作

实现集合 A 与 B 并操作的代码如下：

```
// 集合并操作
if(getOperatorType() == UNION)
{
    if(g1 < 127 && g2 < 127)
        continue;
    output[index] = -1; // make it white
}
```

因为原图像 A 与 B 在绘制时使用了反锯齿边缘，边缘像素没有真正的二值化，所以上面的代码中，假设像素值低于 127 为黑色像素，不做处理。

2. 集合操作：交

集合 A 与 B 的交操作结果可以表示为： $D = A \cap B = \{p | p \in A \text{ and } p \in B\}$ ，在某些情况下集合 A 与 B 可能没有包含共同元素，这时 $A \cap B = \emptyset$ 为空集。假设 A 与 B 是二值图像像素数组，其交集可以表示为 $D = A \text{ and } B$ ，其中像素 $p =$ where (A and B)。假设二值图像 A、B 分别为图 11-1 与图 11-2，则交集操作以后的结果为图 11-4。

实现集合 A 与 B 交操作的代码如下：

```
// 集合交操作
else if(getOperatorType() == INTERSECTION)
{
    if(g1 > 127 && g2 > 127)
        output[index] = -1; // make it white
    else
        continue;
}
```

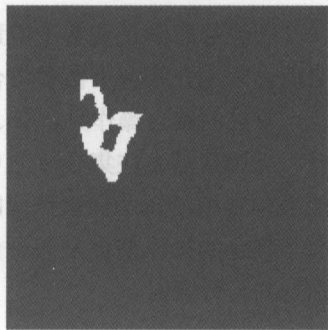


图 11-4 交操作

3. 集合操作：补

假设图 11-1 的全部像素集合为 G ，其中白色对象集合为 A ，则 A 集合的补操作结果为： $A^c = \{w | w \in G \text{ and } w \notin A\}$ ，其中 w 表示为图像像素点。在实际图像处理中，集合补操作可以通过像素取反实现，即 0 取值为 255，255 取为 0 作为结果输出。对图 11-1 进行集合补操作的结果如图 11-5 所示。

这里为了显示需要，把白色像素值从 255 降低到 200。实现集合补的代码如下：

```
// 集合补操作
else if(getOperatorType() == COMPLEMENT)
{
    if(g1 > 127)
    {
        output[index] = -16777216;
    }
    else
    {
        output[index] = (0xff << 24) | (200 << 16) | (200 << 8) | 200;
    }
}
```

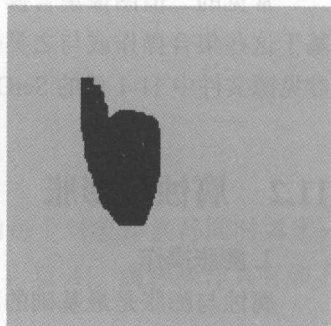


图 11-5 补操作

4. 集合操作：差

假设有集合 A 与 B ，其差操作可以表示为：

$$A - B = \{w | w \in A \text{ and } w \notin B\} = A \cap B^c$$

从上面可以看到，二值图像集合差操作表示 A 与 B 的补的交集，若 A 与 B 分别为图 11-1 与图 11-2，则差操作以后的结果如图 11-6 所示。

实际二值图像 A 与 B 集合差操作的代码如下：

```
// 集合差操作
else if(getOperatorType() == DIFFERENCE)
{
    // 求补
    if(g2 > 127)
    {
        g2 = 0;
    }
    else
    {
        g2 = 255;
    }
    // 求交
    if(g1 > 127 && g2 > 127)
        output[index] = -1; // make it white
    else

```



图 11-6 差操作

```

        continue;
    }
}

```

常见的二值图像集合操作并、交、补、差就介绍到这里，图像形态学的大部分操作都是基于这些集合操作或与之类似的操作完成二值图像处理的。完整的二值图像集合操作源代码参见源文件中 11-1 里的 SetOperatorFilter.java，运行与测试的图像为 A-Set.png 与 B-Set.png。

11.2 腐蚀与膨胀

1. 膨胀操作

腐蚀与膨胀是最基础的图像形态操作，腐蚀与膨胀都是基于集合交操作的，它使用结构元素与图像像素集合做处理得到最终结果。假设 A 为图像像素集合，图像 B 为结构元素，则膨胀操作可以表示为：

$$A \oplus B = \{s | ((\hat{B})_s \cap A) \subseteq A\}$$

也就是说，集合 B 在 A 上移动，得到的交集不为空时，设像素点 s 为对象像素 255，则 A 表示二值图像中白色对象像素点集合。通常图像膨胀操作有如下功能：

- 对象大小增加一个像素。
- 平滑对象边缘。
- 减少或填充对象之间的距离或者对象上小孔。

假设结构元素为 3×3 大小，实现图像膨胀操作的代码如下：

```

// 结构元素宽与高
int seh = this.getStructureElements().length/2;
int sew = this.getStructureElements().length/2;
Arrays.fill(output, -16777216); // black
// 膨胀
for (int row = 0; row < height; row++) {
    for (int col = 0; col < width; col++) {
        index = row * width + col;
        boolean found = false;
        for (int er = -seh; er <= seh; er++) {
            {
                int nrow = row + er;
                for (int ec = -sew; ec <= sew; ec++) {
                    {
                        int ncol = col + ec;
                        int g1 = getPixel(setA, width, height, ncol, nrow);
                        if (g1 > 127) {
                            found = true;
                            break;
                        }
                    }
                }
            }
        }
    }
}

```

```

        if(found)break;
    }
    BoundaryExtractionFilter.java 的源代码参见下文。
}
if(found) { // B与A有交集, 中心像素设为白色
    output[index] = -1; // make it white
}
}
}

```

2. 腐蚀操作

腐蚀操作就是结构元素 B 在图像 A 上移动, 若 B 中对应的每个对象像素都同时属于 A 则保留, 否则丢弃该像素。腐蚀操作可以表示为 $A \ominus B = \{s | (B)_s \subseteq A\}$, 腐蚀操作具有如下功能:

- 对象大小减小一个像素。
- 平滑对象边缘。
- 弱化或分割对象之间的半岛型连接。

假设结构元素为 3×3 大小, 则实现图像腐蚀操作的代码如下:

```

// 结构元素宽与高
int seh = this.getStructureElements().length/2;
int sew = this.getStructureElements().length/2;
Arrays.fill(output, -16777216); // black
// 腐蚀
for (int row = 0; row < height; row++) {
    for (int col = 0; col < width; col++) {
        index = row * width + col;
        // int cg = getPixel(setA, width, height, col, row);
        boolean found = false;
        for(int er=-seh; er<=seh; er++)
        {
            int nrow = row + er;
            for(int ec=-sew; ec<=sew; ec++)
            {
                int ncol = col + ec;
                int g1 = getPixel(setA, width, height, ncol, nrow);
                if(g1<127)
                {
                    found = true;
                    break;
                }
            }
        }
        if(found)break;
    }
}
// 结构元素宽与高
int seh = this.getStructureElements().length/2;
int sew = this.getStructureElements().length/2;
Arrays.fill(output, -16777216); // black
// 腐蚀
for (int row = 0; row < height; row++) {
    for (int col = 0; col < width; col++) {
        index = row * width + col;
        boolean found = false;
        for(int er=-seh; er<=seh; er++)
        {
            int nrow = row + er;
            for(int ec=-sew; ec<=sew; ec++)
            {
                int ncol = col + ec;
                int g1 = getPixel(setA, width, height, ncol, nrow);
                if(g1<127)
                {
                    found = true;
                    break;
                }
            }
        }
        if(!found) { // B与A有交集, 中心像素设为白色
            output[index] = -1; // make it white
        }
    }
}

```

```

    }
}

```

3. 边缘提取

使用膨胀与腐蚀操作实现二值图像中对象边缘提取时，假设 A 为二值图像集合，其中黑色背景像素值为 0，则对象集合 $A = \{p(x,y) \mid p(x,y) \neq 0 \text{ 且 } p(x,y) \in A\}$ 。利用集合操作得到的对象边缘像素集合为 $\beta(A) = A \cap (A \ominus B)^c$ ，其中，结构元素 B 为 3×3 大小。可以看出图像形态学提取二值图像对象边缘就是求得腐蚀操作的结果并取反，之后再与原来的 A 进行集合交操作，其结果即为二值图像中对象边缘。基于上述集合操作原理，实现二值图像中对象边缘提取的代码如下：

```

// 腐蚀操作
BufferedImage erosionImage = super.filter(src, null);
// 获取腐蚀操作之后的像素集合与原像素集合
int[] setA = new int[width*height];
int[] setB = new int[width*height];
int[] output = new int[width*height];
getRGB( src, 0, 0, width, height, setA );
getRGB( erosionImage, 0, 0, width, height, setB );
int index = 0;
// black
Arrays.fill(output, -16777216);
// 提取边缘
for (int row = 0; row < height; row++) {
    for (int col = 0; col < width; col++) {
        index = row * width + col;
        int pa = getPixel(setA, width, height, col, row);
        int pb = getPixel(setB, width, height, col, row);
        if(pa < 127) continue;
        // 对B求补
        if(pb > 127)
        {
            pb = 0;
        }
        else
        {
            pb = 255;
        }

        // 设置边缘像素为白色
        if(pb == 255)
        {
            output[index] = -1;
        }
    }
}

```

实际运行的效果如图 11-7 所示。

在图 11-7 中左边为原图，右边为提取图像边缘之后的结果。完整的二值对象边缘提取类 BoundaryExtractionFilter.java 的源代码参见源文件中的 11-2。

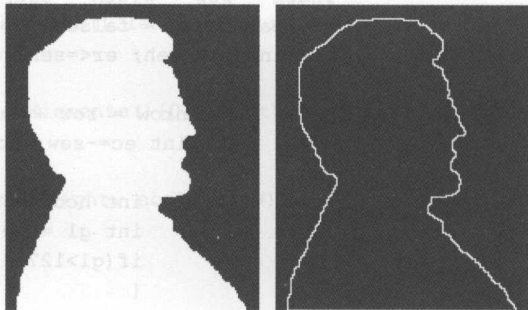


图 11-7 轮廓提取

二值图像中的对象腐蚀可以看成是对背景颜色的膨胀过程，但是腐蚀过程绝对不是膨胀过程的反操作，而且腐蚀与膨胀都可以根据实际需要使用不同的结构元素，从而得到不同的处理结果。此外，进行过腐蚀或膨胀操作的图像可以继续相同的操作，可通过此方法比较多次腐蚀或膨胀与一次腐蚀或膨胀之间的差异，从而更好地理解本节内容。本节中实现膨胀与腐蚀类的源代码参见源文件中 11-2 里的 DilationFilter.java 与 ErosionFilter.java，运用与测试使用的图为图 11-1。

11.3 开闭操作

图像处理中的开闭运算是两个非常重要的数学形态学操作，它们同时都继承自基本的腐蚀与膨胀操作，这些操作一般都会应用在二值图像的分析与处理上。

1. 开操作

开操作有点像腐蚀操作，可用于去除对象像素边缘，但是不会像腐蚀操作一样去除那么多的边缘像素。开操作主要用来去除其他不符合结构元素的前景区域像素，同时保留结构。开操作的定义是一个腐蚀操作再接着一个膨胀操作，两个操作使用相同的结构元素。其中结构元素形状根据开操作的要求不同，结构元素可以是圆形、正方形、矩形等。开操作可以表示为：

$$A \circ B = (A \ominus B) \oplus B$$

即先腐蚀后膨胀，其中 B 为结构元素。实现二值图像开操作的代码如下：

```
// 腐蚀操作
src = super.filter(src, null);
// 获取腐蚀操作之后的像素集合与原像素集合
int[] setA = new int[width*height];
int[] output = new int[width*height];
getRGB( src, 0, 0, width, height, setA );
int index = 0;
// 结构元素宽与高
int seh = this.getStructureElements().length/2;
int sew = this.getStructureElements()[0].length/2;
Arrays.fill(output, -16777216); // black
// 膨胀
for (int row = 0; row < height; row++) {
```



```

for (int col = 0; col < width; col++) {
    index = row * width + col;
    boolean found = false;
    for (int er = -seh; er <= seh; er++)
    {
        int nrow = row + er;
        for (int ec = -sew; ec <= sew; ec++)
        {
            int ncol = col + ec;
            int g1 = getPixel(setA, width, height, ncol, nrow);
            if (g1 > 127)
            {
                found = true;
                break;
            }
        }
        if (found) break;
    }
    if (found) { // B与A有交集, 中心像素设为白色
        output[index] = -1; // make it white
    }
}
}

```

完整的开操作类 `OpenFilter.java` 的源代码参见源文件中的 11-3, 为了减少不必要的代码编写, 这里的开操作类 `OpenFilter` 继承了源文件中 11-2 里的腐蚀类 `ErosionFilter`。图 11-8 是基于不同的结构元素对二值图像处理结果的实例。

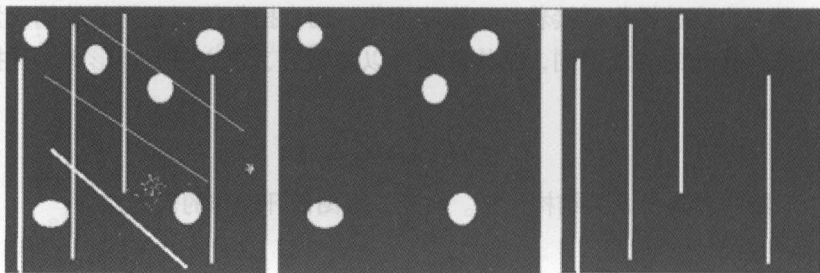


图 11-8 开操作

其中最左边是原二值图像, 中间是基于 5×5 的结构元素开操作结果, 右边是基于 30×2 的结构元素开操作结果。在上述代码中计算结构元素的高与宽时, 若结构元素高度或宽度为 1, 则容易发生误差, 所以更精确地获取结构元素宽与高的代码如下:

```

int seh = (int) (this.getStructureElements().length/2.0f + 0.5f);
int sew = (int) (this.getStructureElements()[0].length/2.0f + 0.5f);

```

根据新的 `seh` 与 `sew` 的值可知, 上述代码中的结构元素循环也需要做出适当的改变, 新

的结构元素循环完成集合操作代码如下:

```
for(int er=-seh; er<(this.getStructureElements().length-seh); er++)
{
    int nrow = row + er;
    for(int ec=-sew; ec<(this.getStructureElements()[0].length - sew); ec++)
    {
        int ncol = col + ec;
        int g1 = getPixel(setA, width, height, ncol, nrow);
        if(g1>127)
        {
            found = true;
            break;
        }
    }
    if(found)break;
}
```

完整的二值图像开操作代码参见源文件中的 `OpenFilter.java` 即可。从上述例子也可以看出图像开操作有根据结构元素选择匹配形状对象功能。

2. 闭操作

闭操作是图像形态学中的重要操作之一, 闭操作也是基于腐蚀与膨胀的基于操作组合而成的, 可以运用于二值图像或灰度图像。闭操作的作用是保留背景区域与结构元素形状相似的像素, 去掉其他不符合的背景区域。闭操作是首先运用结构元素完成膨胀操作, 根据得到的结果再进行腐蚀操作, 可以表示为

$$A \cdot B = (A \oplus B) \ominus B$$

其中 B 表示为结构元素, A 表示二值图像像素集合。实现二值图像闭操作的代码如下:

```
// 膨胀操作
src = super.filter(src, null);
// 获取腐蚀操作之后的像素集合与原像素集合
int[] setA = new int[width*height];
int[] output = new int[width*height];
getRGB( src, 0, 0, width, height, setA );
int index = 0;
// 结构元素宽与高
int seh = (int)(this.getStructureElements().length/2.0f + 0.5f);
int sew = (int)(this.getStructureElements()[0].length/2.0f + 0.5f);
Arrays.fill(output, -16777216); // black
// 腐蚀操作
for (int row = 0; row < height; row++) {
    for (int col = 0; col < width; col++) {
        index = row * width + col;
        boolean found = false;
        for(int er=-seh; er<(this.getStructureElements().length-seh); er++)
        {
            int nrow = row + er;
```

```
for(int ec=-sew; ec<(this.getStructureElements()[0].length - sew);
    ec++)
{
    int ncol = col + ec;
    int gl = getPixel(setA, width, height, ncol, nrow);
    if(gl<127)
    {
        found = true;
        break;
    }
}
if(found)break;

if(!found) { // B与A有交集，中心像素设为白色
    output[index] = -1; // make it white
}
```

为了减少重复代码，闭操作同样继承了前面的 DilationFilter 类。基于闭操作使用 10×10 的结构元素，对 close.png 二值图像处理的结果如图 11-9 所示。

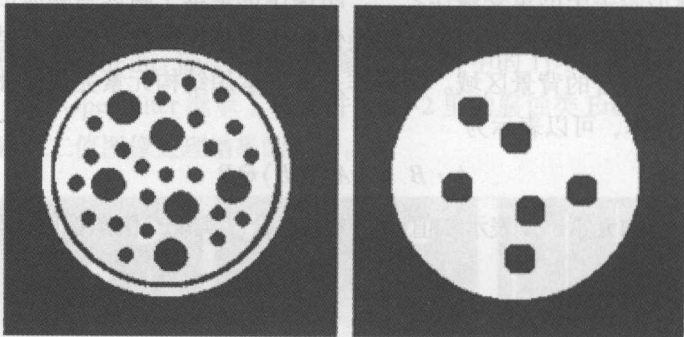


图 11-9 闭操作结果

在图 11-9 中，左边为二值图像，右边为处理以后的结果。完整的闭操作源代码参见源文件中的 CloseFilter.java 即可，闭操作还可以用来选择性填充图像背景区域。

11.4 Hit-and-Miss 变换操作

在图像形态学中，Hit-and-Miss 变换常被用来匹配指定形状的二值图像中的对象，它由一些基本的二值图像操作组合而成，跟腐蚀与膨胀不同的是，Hit-and-Miss 的结构元素中既包括前景像素，也包括背景像素，如果结构元素中没有指明前景像素或背景像素，则表示二者都可以，图 11-10 为

	1	
0	1	1
0	0	

图 11-10 结构元素

一个这样的结构元素示例。

其中 0 表示背景像素黑色, 1 表示前景像素白色, 灰度表示忽略。

1. 操作定义与原理

在涉及具体的编程实现方法, 以及介绍 Hit-and-Miss 变换是由哪些基本图像形态学操作组成的之前, 首先定义一下什么是 Hit, 什么是 Miss。当结构元素 B 在指定的二值图像像素集合上发生重叠时称为 Hit, 当结构元素 B 与指定二值图像像素集合完全没有重叠时称为 Miss, 假设有二值图像集合 A , 结构元素 W , 其中其前景元素集合 B_1 , 背景元素集合 B_2 。则 Hit 操作可以表示为: $A \odot B_1$, Miss 操作则可以表示为: $A^c \odot B_2$, 完整的 Hit-and-Miss 变换表示为: $A \otimes B = (A \odot B_1) \cap [A^c \odot B_2]$

其实质是根据模板的精准匹配, 图 11-10 中的模板可以用来寻找二值区域的左下角像素点。基于图 11-10 实现 Hit-and-Miss 操作得到的结果如图 11-11 所示 (左边为二值图像, 右边是 Hit-and-Miss 操作结果)。

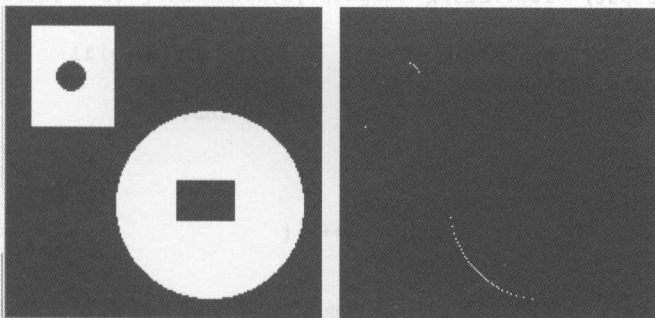


图 11-11 基于结构元素操作之后

2. 代码实现

首先对图像进行二值化处理, 处理以后获取图像像素数据, 归一化为 0 表示黑色背景像素, 1 表示白色对象像素。在定义的结构元素模板中, 0 表示黑色像素, 1 表示白色像素, 2 表示既可以为 0 也可以为 1。本节中使用的模板为图 11-10。完整的 Hit-and-Miss 操作编程实现步骤如下:

- 1) 获取输入图像, 完成二值化, 这里继承了第 6 章的二值图像处理类 BinaryFilter。
- 2) 获取像素并归一化。
- 3) Hit-and-Miss 操作, 发现关键像素点。
- 4) 输出结果图像。

基于上述四步的代码实现片段如下:

```
int width = src.getWidth();
int height = src.getHeight();
if (dest == null)
```

```

dest = createCompatibleDestImage( src, null );

// 二值化
src = super.filter(src, null);
int[] setA = new int[width*height];
int[] output = new int[width*height];

// 像素归一化
getRGB( src, 0, 0, width, height, setA );
for(int i=0; i<setA.length; i++)
{
    int tr = (setA[i] >> 16) & 0xff;
    setA[i] = tr / 255;
}

// 获取中心元素 颜色- 白色为1, 黑色为0
int index = 0;
int total = countZeroAndOne();
Arrays.fill(output, -16777216);

// 结构元素宽与高
int rr = template.length/2;
int rc = template[0].length/2;

// 腐蚀操作, 初始化
for (int row = 0; row < height; row++) {
    for (int col = 0; col < width; col++) {
        index = row * width + col;
        int count = 0;
        for(int trow=-rr; trow<=rr; trow++)
        {
            if((row + trow) < 0 || (row + trow) >= height)
            {
                continue;
            }
            for(int tcol=-rc; tcol<=rc; tcol++)
            {
                if((col + tcol) < 0 || (col + tcol) >= width)
                {
                    continue;
                }
                if(template[trow+rr][tcol+rc] == 2)
                {
                    continue;
                }
                int index2 = (col+tcol) + (row+trow) * width;
                if(setA[index2] == template[trow+rr][tcol+rc])
                {
                    count++;
                }
            }
        }
    }
}

```



```

    }
    if(count == total)
    {
        output[index] = -1;
    }
}

setRGB(dest, 0, 0, width, height, output);
return dest;

```

其中方法 `countZeroAndOne()` 是用来计数模板中前景与背景像素个数的，其实现代码如下：

```

private int countZeroAndOne()
{
    int count = 0;
    for(int i=0; i<template.length; i++)
    {
        for(int j=0; j<template[i].length; j++)
        {
            if(template[i][j] == 1 || template[i][j] == 0)
            {
                count++;
            }
        }
    }
    return count;
}

```

完整的 Hit-and-Miss 变换源代码 `HitAndMissFilter.java` 参见源文件中的 11-4 即可。

11.5 距离变换

1. 距离变换概述

距离变换是二值图像处理与操作中常用的手段，在骨架提取、图像窄化中常有应用。距离变换的结果是得到一张与输入二值图像类似的灰度图像，但是灰度值只出现在前景区域，并且越远离背景边缘的像素灰度值越大。根据度量距离的方法不同，距离变换的结果稍有不同，假设两个像素点 $p_1(x_1, y_1)$ 与 $p_2(x_2, y_2)$ ，常见的距离计算方法有如下几种。

- 欧几里得距离公式：Distance = $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$
- 曼哈顿距离公式：Distance = $|x_1 - x_2| + |y_1 - y_2|$
- 象棋格距离公式：Distance = $\max(|x_1 - x_2|, |y_1 - y_2|)$

一旦选择了距离度量公式，就可以在二值图像的距离变换中使用了。二值图像的距离变换方法有很多，这里基于形态学腐蚀操作来实现距离变换，主要是利用腐蚀操作的特点每次腐蚀一个像素前景边缘像素，从而最终得到基于距离的像素梯度，腐蚀操作的停止条件是所

有前景像素都被完全腐蚀。这样根据腐蚀的先后顺序，我们就得到了各个前景像素点到前景中心骨架像素点的距离。将各个像素点的距离值设置为不同的灰度值，这样就完成了二值图像的距离变换。通常腐蚀操作可以支持各种结构元素，而对于通过腐蚀实现距离变换来说，最好的结构元素就是 3×3 大小的窗口模板。

2. 代码实现 (基于腐蚀)

基于形态学腐蚀操作实现距离变换操作的代码实现大致可以分为如下几步：

- 1) 二值图像灰度值初始化，全部像素点初始化灰度值为 0。
- 2) 基于腐蚀生成图像前景边缘集合。
- 3) 基于前景边缘腐蚀生成背景边缘集合。
- 4) 迭代对每个前景边缘像素计算距离，然后放入背景边缘集合中，检查背景边缘中的每个像素点的八邻域像素点，如果发现有前景像素点，则加到前景边缘像素集合中。
- 5) 直到前景像素为空，才停止迭代。
- 6) 根据每个像素点的距离值设置灰度值大小，输出距离变换之后的图像。

基于上述步骤，初始化部分的代码如下：

```
this.scaleValue = scaleValue;
this.offsetValue = offsetValue;
this.inputImage = src;
this.width = src.getWidth();
this.height = src.getHeight();
int[] inPixels = new int[width*height];
getRGB( src, 0, 0, width, height, inPixels );
int index = 0;
pixels2D = new int[height][width]; // row, column
greyLevel = new int[height][width];
for(int row=0; row < height; row++)
{
    for(int col=0; col<width; col++)
    {
        index = row * width + col;
        int grayValue = (inPixels[index] >> 16) & 0xff;
        pixels2D[row][col] = grayValue;
        greyLevel[row][col] = 0;
    }
}

generateForegroundEdge();
generateBackgroundEdgeFromForegroundEdge();
```

其中，scaleValue 与 offsetValue 是用来调节距离变换以后的图像灰度值的。实现距离变换部分的代码如下所示：

```
// calculate the distance here!!
int index = 1;
```

```
while (foregroundEdgePixels.size() > 0) {
```

```
distanceSingleIteration(index);
```

```
++index;
```

}

其中，方法 `distanceSingleIteration()` 的代码如下：

```
Iterator<Point> localIterator = foregroundEdgePixels.iterator();
```

```
while (localIterator.hasNext()) {
```

```
Point localPoint = new Point((Point)localIterator.next());
```

```
backgroundEdgePixels.add(localPoint);
```

```
removePixel(localPoint);
```

```
greyLevel[localPoint.y][localPoint.x] = paramInt;
```

}

```
generateForegroundEdgeFromBackgroundEdge();
```

根据距离变换结果调节灰度值与输出结果的代码如下:

```
// loop the each pixel and assign the color value according to distance value
```

```
for (int row = 0; row < inputImage.getHeight(); row++) {
```

```
for (int col = 0; col < inputImage.getWidth(); col++) {
```

```
if (greyLevel[row][col] > 0) {
```

```
int colorValue = (int)Math.round(greyLevel[row][col] *
    scaleValue + offsetValue);
```

```
colorValue = colorValue > 255 ? 255 : ((colorValue < 0) ? 0
: colorValue);
```

```
this.pixels2D[row][col] = colorValue;
```

1

```
// build the result pixel data at here !!!
```

```
if ( dest == null )
```

```
dest = createCompatibleDestImage(inputImage, null);
```

```
index = 0;
```

```
int[] outPixels = new int[width*height]:
```

```
for(int row=0; row<height; row++) {
```

```
int ta = 0, tr = 0, tq = 0, tb = 0;
```

```
for(int col=0; col<width; col++)
```

```
index = row * width + col;
```

```
tr = tg = tb = this.pixels2D[row][col];
```

```
ta = 255;
```

```
outPixels[index] = (ta << 24) | (tr << 16) | (tg << 8) | tb;
```

}

```
setRGB( dest, 0, 0, width, height, outPixels );
```

```
return dest;
```

完整的距离变换源代码参见源文件中 11-5 的 DistanceTransform.java 即可。二值图像的距离

离变换实现方法还有很多，另外一种比较常见的算法为 Chamfer Distance Transform(CDT)，感兴趣的读者可以自己进一步阅读与研究。

11.6 分水岭算法

1. 定义与概述

图像分水岭变换的定义是基于图像像素的灰度值距离来说的，根据选取的距离功能不同，得到的结果稍有不同。常见的分水岭变换分为基于拓扑距离与基于浸泡理论，基于拓扑距离的分水岭算法涉及更多的图的算法知识，而浸泡理论的分水岭算法的定义相对简单，主要基于形态学知识。所以结合本章内容，主要介绍基于浸泡理论的分水岭算法定义。

基于浸泡理论实现分水岭变换算法是由 Vincent 与 Soille 在 1991 提出的，算法思想是假设灰度图像 D ，其最小与最大灰度值分别为 h_{\min} 与 h_{\max} ，且灰度值从最小值 h_{\min} 开始，浸泡灰度值最小的像素点，逐步升高灰度值实现盆地扩展，对任意一个灰度值像素，如果不是现有盆地的扩展就是一个新的盆地，或者是个分水岭 W 。基于递归完成对所有灰度值处理之后就得到了最终分水岭像素点集合：

$$Wshed(f) = D/X_{h_{\max}}$$

基于四邻域像素举例说明如图 11-12 所示。

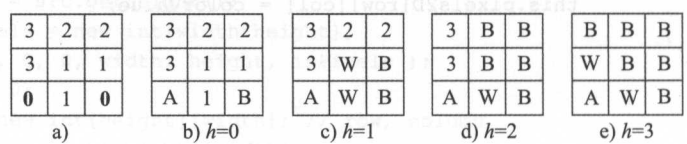


图 11-12 分水岭过程

其中 a 表示输入 3×3 的像素灰度值，粗体部分表示最小灰度值 0，b 表示 $h = 0$ 时发现 A 与 B 两个盆地，c 表示 $h = 1$ 时得到盆地 B 扩展与两个被标记为分水岭的像素 W，d ~ e 时，分水岭像素点 $W(1,1)$ 被更新并得到最终的两个分水岭像素 W。

2. 步骤与编码实现

浸泡算法的分水岭变换是基于队列的，首先要对所有像素进行初始化，假设任意一个像素可以有四种状态，分别为初始化 (INIT)、未处理 (MASK)、被标记为分水岭 (Wshed)、被标记为 Label。

完成所有像素的初始化以后，对每个像素完成八邻域寻找。然后从灰度 0 开始，步长为 1，逐渐过渡到 255 进行如下几步处理：

- 1) 初始化所有灰度值等于给定灰度 h 的像素点，设置为 MASK，如果邻域像素点被标记或为分水岭像素，则将该像素点添加到队列中。
- 2) 如果队列不为空，出列队列中每个像素，根据条件处理开始扩展盆地。

3) 使用 DFS 算法, 对于给定灰度 h 的像素点, 如果状态仍然是 MASK, 则标记新盆地。

4) 根据上述处理结果, 如果最终像素点被标记为分水岭像素 (WSHED), 则显示为白色, 得到分水岭连接线。

基于上述几步, 浸泡实现图像分水岭变换的代码片段如下:

```
int width = src.getWidth();
int height = src.getHeight();
if ( dest == null )
    dest = createCompatibleDestImage( src, null );
int[] input = new int[width*height];
getRGB( src, 0, 0, width, height, input );
int index = 0;
// 初始化每个像素值
Map<String, PixelPoint> pixelMap = new HashMap<String, PixelPoint>();
// 直方图高度
Map<Integer, List<PixelPoint>> heightMap = new HashMap<Integer, List<PixelPoint>>();
for (int row = 0; row < height; row++) {
    int tr=0;
    for (int col = 0; col < width; col++) {
        index = row * width + col;
        tr = (input[index] >> 16) & 0xff;
        PixelPoint pp = new PixelPoint(row, col, tr);
        pixelMap.put((row + "," + col) , pp);
        if(heightMap.get(Integer.valueOf(tr)) == null)
        {
            heightMap.put(Integer.valueOf(tr), new ArrayList<PixelPoint>());
        }
        heightMap.get(Integer.valueOf(tr)).add(pp);
    }
}
// 八邻域链接像素寻找
for (int row = 0; row < height; row++) {
    for (int col = 0; col < width; col++) {
        index = row * width + col;
        PixelPoint cpp = pixelMap.get(row + "," + col);
        for(int nr=-1; nr<2; nr++)
        {
            if((nr+row) < 0 || (row+nr) >= height)
                continue;
            for(int nc=-1; nc<2; nc++)
            {
                if((nc+col) < 0 || (nc+col) >=width)
                    continue;
                int index2 = (row+nr) * width + nc + col;
                if(index == index2) continue; // skip it
                cpp.getNeighbours().add(pixelMap.get((row+nr) + "," + (nc
                    + col)));
            }
        }
    }
}
```

```

    }
}
// 初始化浸泡算法
FIFOQueue myQueue = new FIFOQueue();
int curLab = 0;
int curdist = 0;
int _watershedPixelCount = 0;
// 开始浸泡
for(int h=0; h<256; h++)
{
    if(heightMap.get(Integer.valueOf(h)) == null) continue;
    for(PixelPoint pp : heightMap.get(Integer.valueOf(h)))
    {
        pp.setLabelToMASK();
        for(PixelPoint neighbourPixel : pp.getNeighbours())
        {
            if(neighbourPixel.getLabel() >= 0) { // water-shed or tag label
                pp.setDistance(1);
                myQueue.fifo_add(pp);
                break;
            }
        }
        curdist = 1;
        myQueue.fifo_add_FICTITIOUS();
        // 扩展盆地
        while(true)
        {
            PixelPoint p = myQueue.fifo_remove();
            if(p.isFictitious())
            {
                if(myQueue.fifo_empty())
                {
                    break;
                }
            }
            else
            {
                myQueue.fifo_add_FICTITIOUS();
                curdist++;
                p = myQueue.fifo_remove();
            }
        }
        for(PixelPoint q : p.getNeighbours())
        {
            if(q.getDistance() <= curdist && (q.getLabel() > 0 ||
                q.isLabelWSHED()))
            // if(q.getDistance() <= curdist && (q.getLabel() >= 0))
            {
                if(q.getLabel() > 0)
            {

```

```

        if(p.isLabelMASK())
        {
            p.setLabel(q.getLabel());
        }
        else if(p.getLabel() != q.getLabel())
        {
            p.setLabelToWSHED();
            _watershedPixelCount++;
        }
    }
    else if(p.isLabelMASK())
    {
        p.setLabelToWSHED();
        _watershedPixelCount++;
    }
    else if(q.isLabelMASK() && q.getDistance() == 0)
    {
        q.setDistance(curdist+1);
        myQueue.fifo_add(q);
    }
}

// DFS - tag label for all mask pixel point
if(heightMap.get(Integer.valueOf(h)) == null) continue;
for(PixelPoint maskPxieiPoint : heightMap.get(Integer.valueOf(h)))
{
    // reset distance to zero
    maskPxieiPoint.setDistance(0);
    if(maskPxieiPoint.isLabelMASK()) // new minimum region
    {
        curlabel++;
        myQueue.fifo_add(maskPxieiPoint);
        maskPxieiPoint.setLabel(curlabel);
        // 组件标记算法
        while(!myQueue.fifo_empty())
        {
            PixelPoint q = myQueue.fifo_remove();
            for(PixelPoint qn : q.getNeighbours())
            {
                if(qn.isLabelMASK())
                {
                    myQueue.fifo_add(qn);
                    qn.setLabel(curlabel);
                }
            }
        }
    }
}

```



```

}
// 输出统计
if(_watershedPixelCount > 0)
{
    System.out.println(" total watershed pixel count = " + _watershedPixelCount);
}
// 显示分水岭线
for (int row = 0; row < height; row++) {
    for (int col = 0; col < width; col++) {
        index = row * width + col;
        PixelPoint pp = pixelMap.get(row + "," + col);
        if(pp.isLabelWSHED() && !pp.allNeighboursAreWSHED())
        {
            input[index] = (255 << 24) | (255 << 16) | (255 << 8) | 255;
        }
    }
}

```

其中用到的类 PixelPoint 是实现每个像素点信息记录的数据结构，FIFOQueue 是基于 LinkedList 的队列。完整的水分水岭浸泡算法源代码参见源文件中的 11-6，上述源代码中给出的是基于八邻域的寻找实现，读者可以自己尝试改成四邻域寻找实现。

11.7 灰度图像腐蚀与膨胀

本章前面已经介绍的形态学知识除了浸泡分水岭算法，其他都是针对二值图像进行处理的，其实很多时候这些形态学知识还被扩展到灰度图像上，实现灰度图像有很多基本的形态学操作，如腐蚀、膨胀、开闭操作等。灰度图像的形态学操作具有实际应用意义，在图像分割、重建等方面均有使用。本节通过介绍灰度图像形态学基本操作——腐蚀与膨胀，帮助读者进一步拓宽与加深对图像形态学处理的认知与理解。

1. 结构元素

对比于二值图像，灰度图像形态操作从像素值的变化方法来说可以分为两类，其中一类为结构元素中像素值固定不变而且相互相等，我们称之为平坦结构元素；另外一类可称为非平坦结构元素，其结构元素中灰度值是连续变化、相互不同的。本节中如果没有特别说明，其结构元素都基于平坦结构元素完成灰度图像腐蚀与膨胀。

2. 腐蚀

假设图像 $f(x, y)$ 是灰度图像，平坦结构元素 $b(x, y)$ 则基于平坦结构元素腐蚀，可以表示为在任意像素点 $p(x, y)$ 结构元素与灰度图像重叠的最小值，公式表示如下：

$$[f \ominus b](x, y) = \min_{(s, t) \in b} \{f(x + s, y + t)\}$$

随着 x 与 y 变化访问灰度图像中的每个像素，最终得到腐蚀之后的结果。非平坦结构元

素 b_N 的灰度图像腐蚀与此类似, 唯一不同的是需要对应减去结构元素中的灰度值得到最小值, 其公式表示如下:

$$[f \ominus b_N](x, y) = \min_{(s, t) \in b_N} \{f(x + s, y + t) - b_N(s, t)\}$$

根据上述定义, 基于平坦结构元素实现灰度图像腐蚀的代码实现如下:

```
int width = src.getWidth();
int height = src.getHeight();

if (dest == null)
    dest = createCompatibleDestImage(src, null);

int[] inPixels = new int[width * height];
int[] outPixels = new int[width * height];
getRGB(src, 0, 0, width, height, inPixels);
int index = 0;
int s = elements[0].length;
int t = elements.length;
int min = 255;
for (int row = 0; row < height; row++) {
    int tg = 0;
    for (int col = 0; col < width; col++) {
        index = row * width + col;
        min = 255;
        for (int subRow = 0; subRow < t; subRow++) {
            int nrow = row + subRow;
            if (nrow < 0 || nrow >= height)
                nrow = 0;
            for (int subCol = 0; subCol < s; subCol++) {
                int ncol = col + subCol;
                if (ncol < 0 || ncol >= width)
                    ncol = 0;
                int index1 = nrow * width + ncol;
                tg = (inPixels[index1] >> 8) & 0xff;
                min = Math.min(tg, min);
            }
        }
        outPixels[index] = (255 << 24) | (min << 16) | (min << 8) | min;
    }
}

setRGB(dest, 0, 0, width, height, outPixels);
return dest;
```

完整的基于平坦结构元素实现灰度图像腐蚀的源代码参见源文件中 11-7 的 GrayErosion-

Filter 类。非平坦结构元素的代码请读者自己根据上述程序改动得到。

3. 膨胀

灰度图像的膨胀操作与腐蚀操作类似，基于平坦结构元素实现灰度图像膨胀操作的公式表示如下：

$$[f \oplus b](x, y) = \max_{(s, t) \in b} \{f(x - s, y - t)\}$$

类似地，基于非平坦结构元素实现灰度图像膨胀操作的公式表示如下：

$$[f \oplus b_N](x, y) = \max_{(s, t) \in b_N} \{f(x - s, y - t) + b_N(s, t)\}$$

通常情况下非平坦结构元素很少使用，从定义可以看出，膨胀以后的灰度图像亮度应该比原图像更亮，而腐蚀之后的灰度图像比原图像更暗。基于平坦结构元素实现灰度图像膨胀的代码如下：

```
int width = src.getWidth();
int height = src.getHeight();

if (dest == null)
    dest = createCompatibleDestImage(src, null);

int[] inPixels = new int[width * height];
int[] outPixels = new int[width * height];
getRGB(src, 0, 0, width, height, inPixels);
int index = 0;
int s = elements[0].length;
int t = elements.length;
int max = 0;
for (int row = 0; row < height; row++) {
    int tg = 0;
    for (int col = 0; col < width; col++) {
        index = row * width + col;
        max = 0;
        for (int subRow = 0; subRow < t; subRow++) {
            {
                int nrow = row - subRow;
                if (nrow < 0 || nrow >= height)
                {
                    nrow = 0;
                }
                for (int subCol = 0; subCol < s; subCol++)
                {
                    int ncol = col - subCol;
                    if (ncol < 0 || ncol >= width)
                    {
                        ncol = 0;
                    }
                    int index1 = nrow * width + ncol;
                    tg = (inPixels[index1] >> 8) & 0xff;
                }
            }
        }
    }
}
```

```
return dest;
```

再和相似度进行古升级方法, 最终得到相似度的结果。最常见的是基于图像分水岭算法实现

11.8 小结

了灰度图像的腐蚀与膨胀操作。

步学习图像形态学的其他知识。

的一部分，只有不断地动手实现，对照理论进行学习，才能真正做到学以致用。

Filter 类。非平面结构元素的代码请读者自己想想！读者中或许得到。

3. 膨胀

灰度膨胀操作是指将输入图像中的每个像素与其邻域中的像素进行比较，如果邻域中的像素值大于或等于输入像素值，则将输入像素值替换为邻域中的最大值。膨胀操作的公式表示如下：



Chapter 12

第 12 章

图像分割

本章将为读者介绍图像分割 (Image Segmentation) 的基本概念与知识，由于图像分割本身涉及很多的数学知识，所以在本章会尽量用大家都能理解的语言组织与介绍这些数学知识。图像分割在对象识别与跟踪、图像搜索、图像编辑、内容压缩等方面都具有十分重要的意义。

常见的图像分割是指根据特定分类方法将图像中的相同对象区别开来，或者从图像背景中分割出图像前景对象。其算法大致可以分为两类，一类是可以自动识别对象分割，整个处理过程不需要人为干预，另外一类是要根据输入参数确定图像中的对象分割。根据图像的不同特征又可以将图像分割法分为三类，分别基于图的知识实现、基于数学统计分析知识实现和基于图像不同区域合并实现。本章对上述几种方法均有涉及，读者也可以自己一步阅读了解。

12.1 抠图真的这么难吗

很多人对 PS 中各种抠图实现移花接木的技术充满兴趣，其实只要学习好本章知识，你就会发现实现类似的抠图应用真的不是想象中的那么难。本质上市场上五花八门的抠图应用都是基于一些常见的图像分割算法，然后再加上点人工操作就完成了。其效果是否理想，除了图像本身的质量以外，其背后还是图像分割算法是个决定性因素之一。

1. 基于统计学知识图像分割方法

一幅图像通过适合的阈值可以分为前景与背景像素，一个极端的例子是图像二值化，但是如果只有一个全局阈值，往往得到的图像分割结果并不理想。常见的都是基于可变阈值来实现图像分割的，基于阈值的图像分割方法要求图像前景对象与背景对象分布融合度不高，

如果背景对象与前景对象相互重叠,则该方法得到的结果会很不理想。

另外一种技术是 Cluster 分类,基于 Cluster 实现图像分割方法是一种非自我监督的图像分割技术,在该方法中,每个 Cluster 集合内部的像素都高度相似,每个没有被指派到特定 Cluster 的像素都要计算出它与每个 cluster 的相似度,从而保证它被放入与之最相似的 Cluster 中,通过不断更新 Cluster 中心像素迭代实现图像分割,基于该技术最常见的算法有 K-Means 算法与 Fuzzy C-Means 算法。

2. 基于区域的图像分割方法

基于区域的图像分割方法,首先通过处理分类获得图像的每个区域,然后根据每个区域的相似度进行合并或分离,最终得到图像分割的结果。最常见的是基于图像分水岭算法实现区域分割,然后根据区域之间的相似度进行区域合并,得到最终的图像分割结果。本章将对此进行详细介绍;另外一种区域图像分割方法是基于图或树的数据结构分割与合并实现。

3. 基于边缘的图像分割方法

最常用的基于边缘的图像分割方法是基于图像一阶或二阶导数计算结果实现的,通常一阶导数求得图像梯度实现分割,二阶导数则通过拉普拉斯算子实现分割。最常用的边缘算子有 Sobel 算子、Prewitt 算子与 Roberts 算子,Canny 边缘提取算法基于高斯模糊实现,可以更好地获取图像边缘。但是在实际操作中,这些边缘操作算子得到的图像边缘最终不能形成闭合区域,无法最终确定图像分割区域,所以还需要通过其他图像处理手段来完成最终的图像分割。

另外,基于灰度图像形态学操作,也常常用来作为图像分割的手段,在某些情况下效果还不错,但是大多数时候,需要进一步合并分割区域。

从上面的介绍也可以看出,图像分割的大致方向就是运用各种数学知识实现对图像每个像素类别的划分。除了上述介绍的这些方法,在应用中经常用到的图像分割方法还有基于高斯混合模型(GMM)方法、基于图的最小割的方法等。

图像分割(Image Segmentation)在对象跟踪、物体识别、匹配中是常见且非常重要的处理算法,纯粹的抠图通常还有另外一个名称,即 GrabCut。这里只是介绍概念,不做进一步的深入探讨,后面将对本节提及到各种图像分割方法给出具体的算法细节与编码实现,帮助读者真正做到理论与实践相结合。

12.2 基于 Mean-Shift 的图像分割

Mean-Shift 算法最早是在 1975 年由 Fukunaga 和 Hostetler 两位提出并实现的。起初并没有引起足够重视,直到 2000 年左右有人将 Mean-Shift 算法用来实现视频中的标记对象跟踪,Mean-Shift 算法的威力才得到了最大限度的释放。本节在通过 Mean-Shift 算法实现图像自动分割时,不需要输入 Cluster 的数目,这与稍后介绍的 K-Means 算法略有不同,决定 Mean-

Shift 算法分类 Cluster 数目的有两个参数, 分别为空间距离 (Radius) 与颜色值的范围 (Color Distance), 假设空间距离 (Radius) 为 3, 表示像素距离在 3 个像素以内, 颜色值范围 (Color Distance) 为 25, 表示颜色差值在 25 以内。

1. 算法解释

本质上 Mean-Shift 算法是一种根据分布密度来决定分类的算法, 假设有图 12-1 所示的左边的像素点分布。

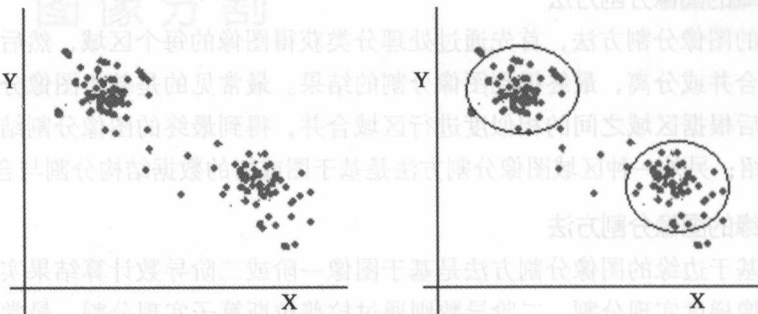


图 12-1 像素点分布

根据图 12-1 中的分布密度, Mean-Shift 算法发现有两个最大密度分布, 如图 12-1 右侧所示。Mean-Shift Cluster 算法假设在 d 维空间 R^d 上有 n 个数据点 x_i , 其中 i 取值范围为 $1 \sim n$, 其多元核密度估算公式表示为:

$$f(x) = \frac{1}{nh^d} \sum_{i=1}^n K\left(\frac{x - x_i}{h}\right)$$

其中 $K(x)$ 表示核, h 为窗口半径大小。对于径向对称核函数, 定义其核 $k(x)$ 满足

$$K(x) = c_{k,d} k(\|x\|^2)$$

其中 $c_{k,d}$ 是归一化常量, 保证 $K(x)$ 整合为 1。最终密度模型在梯度变化 $\nabla f(x) = 0$ 处停止。Mean-Shift 算法就是通过不断根据计算出来的梯度差值来实现窗口移动的, 而且窗口移动的最终方向为概率密度最大点处。关于 Mean-Shift 算法的收敛性证明读者可以查阅相关资料, 这里不再介绍。

对于空间分布的点, Mean-Shift 可以根据密度分布实现分类, 对于图像像素来说, 每个像素点的位置是均匀分布的, 但是别忘记每个像素值都不是均匀分布的, 所以在像素值空间与像素坐标空间结合之后的 5 维空间中 (x, y, r, g, b) 中, 空间距离越近, RGB 值越相似的像素点则越容易被归属到同一个密度分布中, 所以针对这 5 维空间中的每个像素点, 运用 Mean-Shift 直到收敛, 就会得到几个本地最大的密度分布点, 而在 Mean-Shift 算法计算这些最大密度分布点的过程中, $\nabla f(x) \neq 0$ 时位移走过那些像素, 很自然被分到不同本地密度最大点的集合中, 待所有像素都被分类以后, 也就完成了图像的 Mean-Shift 分割。当然这只是理论上的, 实际编码中还要考虑根据输入阈值把最小分类合并。所以, 这里基于 Mean-Shift 算法实现的

图像分割步骤如下:

- 1) 读取像素数组, 将像素值从 RGB 空间转换到 YIQ 空间。
- 2) 开始 Mean-Shift 算法。
 - a) 随机初始化 Mean-Shift 算法开始点 (这样做好处是可以减少计算量)。
 - b) 对每个随机 Mean-Shift 开始点开始 Mean-Shift 操作, 直到收敛。
 - c) 对没有被 Mean-Shift 的像素点, 根据像素值分派到相似的 Cluster 中。
 - d) 根据 Mean-Shift Cluster 中心像素值更新 Cluster 中的每个像素值。
 - e) 循环 a ~ d 直到符合条件退出。
- 3) 合并小于阈值数目的分类。
- 4) 将像素值重新转换到 RGB, 以 Mean-Shift 分类计算得到的值作为分类以后各个像素点值。
- 5) 输出显示图像。



注意 距离计算采用的公式是基于欧几里得距离, 在第二步的终止条件中, 为了避免产生过多计算与无限循环, 人为设置终止条件是重复计算 60 次, 当然可以根据实际情况灵活调整。此外输入参数对 Mean-Shift 图像的 Segmentation 结果也有很大影响, 具体来说, 若参数过大, 会导致图像被过度平滑, 可能只会有一个分类; 若参数过小, 则可能导致图像过度 Segmentation, 即图像分割过多。

2. 实现

主要的 Mean-Shift 算法实现可以看成两大部分, 第一部分是 Mean-Shift 算法运用 Cluster 实现寻找, 第二部分是其后的处理, 主要是实现 Cluster 的合并与结果显示, 这里不再给出具体代码片段, 可以直接参见源文件中的 12-2。

首先, 预处理中图像像素值从 RGB 空间到 YIQ 空间转换的实现代码如下:

```
// convert RGB color space to YIQ color space
float[][] pixelsf = new float[width*height][4];
for(int i=0; i<inPixels.length; i++) {
    int argb = inPixels[i];
    int r = (argb >> 16) & 0xff;
    int g = (argb >> 8) & 0xff;
    int b = (argb) & 0xff;
    pixelsf[i][0] = 0.299f * r + 0.587f * g + 0.114f * b; // Y
    pixelsf[i][1] = 0.5957f * r - 0.2744f * g - 0.3212f * b; // I
    pixelsf[i][2] = 0.2114f * r - 0.5226f * g + 0.3111f * b; // Q
    pixelsf[i][3] = 0.0f; // flag
}
```

随机初始化 Mean-Shift 开始点的实现代码如下:

```
// initialize the centers
for(int i=0; i<numOfCenters; i++)
```

```

{
    int px = random.nextInt(width);
    int py = random.nextInt(height);
    int pIndex = py * width + px;
    meanpoints[i] = new MeanPoint(py, px, new float[]{pixelsf[pIndex][0],
        pixelsf[pIndex][1], pixelsf[pIndex][2]});
}

```

每个中心点实现 Mean-Shift 计算，找出最大密度分布点的实现代码如下：

```

private void meanShfit(MeanPoint meanPoint, Map<MeanPoint, List<PixelPoint>>
    result2, int width, int height, float[][] pixelsf) {
    double shift = 0.0;
    // space distance and color distance
    float radius2 = radius * radius;
    float dis2 = colorDistance * colorDistance;
    // current shift center
    int xc = meanPoint.getCol();
    int yc = meanPoint.getRow();
    float Yc = meanPoint.getRgb()[0];
    float Ic = meanPoint.getRgb()[1];
    float Qc = meanPoint.getRgb()[2];
    // save previous shift center
    int xcOld = meanPoint.getCol();
    int ycOld = meanPoint.getRow();
    float YcOld = meanPoint.getRgb()[0];
    float IcOld = meanPoint.getRgb()[1];
    float QcOld = meanPoint.getRgb()[2];
    // calculate 5D, x, y, YIQ
    float[] yiq = new float[]{Yc, Ic, Qc};
    List<PixelPoint> results = new ArrayList<PixelPoint>();
    do
    {
        xcOld = xc;
        ycOld = yc;
        YcOld = Yc;
        IcOld = Ic;
        QcOld = Qc;

        float mx = 0;
        float my = 0;
        float mY = 0;
        float mI = 0;
        float mQ = 0;
        int num=0;
        // calculate the sum based on generated pixel
        for (int ry=-radius; ry <= radius; ry++) {
            int y2 = yc + ry;
            if (y2 >= 0 && y2 < height) {
                for (int rx=-radius; rx <= radius; rx++) {
                    int x2 = xc + rx;

```

```

if (x2 >= 0 && x2 < width) {
    if (ry*ry + rx*rx <= radius2) {
        yiq = pixelsf[y2*width + x2];
        float Y2 = yiq[0];
        float I2 = yiq[1];
        float Q2 = yiq[2];

        float dY = Yc - Y2;
        float dI = Ic - I2;
        float dQ = Qc - Q2;

        if (dY*dY+dI*dI+dQ*dQ <= dis2) {
            PixelPoint f = new
                PixelPoint(y2, x2);
            f.setRGB(yiq);
            results.add(f);
            yiq[3] = 100; // flag it
            mx += x2;
            my += y2;
            mY += Y2;
            mI += I2;
            mQ += Q2;
            num++;
        }
    }
}

```

12.3 基于 K-Means 的图像分割

K-Means 算法的起源可以追溯到 1957 年，K-Means 算法由 MacQueen 提出。K-Means 算法作为一种聚类算法，因为具有简单、快速、收敛性好等优点，被广泛应用于图像分割、数据挖掘、模式识别等领域。K-Means 算法的基本思想是：预先知道数据集中有多少个 Clusters，这个与 Mean-Shift 有所不同。

1. 算法解释

```

// calculate means
float num_ = 1f/num;
Yc = mY*num_; // 得到平均值
Ic = mI*num_;
Qc = mQ*num_;

1) 假设数据点
2) 根据公式
xc = (int) (mx*num_+0.5);
yc = (int) (my*num_+0.5);
// calculate offset
int dx = xc-xcOld;
int dy = yc-ycOld;
float dY = Yc-YcOld;
float dI = Ic-IcOld;
float dQ = Qc-QcOld;
// shift
shift = dx*dx+dy*dy+dY*dY+dI*dI+dQ*dQ;
// update center location and YIQ value
meanPoint.getRgb()[0] = Yc;
meanPoint.getRgb()[1] = Ic;
meanPoint.getRgb()[2] = Qc;
meanPoint.setCol(xc);
meanPoint.setRow(yc);

```

第三步中，还可以使用欧氏距离或曼哈顿距离。介绍了 K-Means 算法运用到图像分割中。图像本质上是像素点的集合，每个像素点都有 RGB 值。假设全部的像素数据点有 K 个 Cluster，中心点的 RGB 空间距离，与其中一个 Cluster 中心点距离最近，对全部的像素点执行同样的操作，直到收敛就完

```

    }
    while(shift>0.1);
    // start to merge the features, find the local maximum
    boolean flag = false;
    for(MeanPoint mpKey : result2.keySet())
    {
        int deltaY = meanPoint.getRow() - mpKey.getRow();
        int deltaX = meanPoint.getCol() - mpKey.getCol();
        float deltaYc = mpKey.getRgb()[0] - meanPoint.getRgb()[0];
        float deltaIc = mpKey.getRgb()[1] - meanPoint.getRgb()[1];
        float deltaQc = mpKey.getRgb()[2] - meanPoint.getRgb()[2];
        float twoSpaceDis = deltaY * deltaY + deltaX * deltaX;
        float twoColorDis = deltaYc * deltaYc + deltaIc * deltaIc + deltaQc *
            deltaQc;
        if(twoSpaceDis <= radius2 && twoColorDis <= dis2)
        {
            List<PixelPoint> pList = result2.get(mpKey);
            MergeTwo(pList, results);
            flag = true;
            break;
        }
    }
    // new density center
    if(!flag)
    {
        result2.put(meanPoint, results);
    }
}

```

合并 Cluster 与更新像素值的代码如下：

```

// now assign remaining pixels to feature space
for(int row=0; row<height; row++) {
    for(int col=0; col<width; col++) {
        index = row * width + col;
        if(pixelsf[index][3] == 0.0f)
        {
            MeanPoint smp = findSimilarMeans(pixelsf[index], allPoints.
                keySet());
            PixelPoint f = new PixelPoint(row, col);
            f.setRGB(new float[]{pixelsf[index][0], pixelsf[index]
                [1], pixelsf[index][2]});
            allPoints.get(smp).add(f);
        }
        else
        {
            pixelsf[index][3] = 0.0f; // reset
        }
    }
}
}

```

```

// update with means centers
for(MeanPoint mpKey : allPoints.keySet())
{
    for(PixelPoint ft : allPoints.get(mpKey))
    {
        index = ft.getRow() * width + ft.getCol();
        pixelsf[index][0] = mpKey.getRgb()[0];
        pixelsf[index][1] = mpKey.getRgb()[1];
        pixelsf[index][2] = mpKey.getRgb()[2];
    }
}

```

完整的基于 Mean-Shift 算法实现图像分割的源代码参见源文件中 12-2 的 MeanShiftAlgo.java, 其中用到的两个数据结构类 PixelPoint.java 与 MeanPoint.java 只是用来存储像素点位置信息与像素值的。读者可以进一步优化代码实现, 完成自己版本的 Mean-Shift 图像分割算法。

另外常见的 Mean Shift 图像分割多是基于 LUV 色彩空间的, 读者也可以在理解本节内容的基础上进一步改写程序, 完成基于 LUV 色彩空间的 Mean-Shift 图像分割代码。

12.3 基于 K-Means 的图像分割

K-Means 算法的起源可以追溯到 1957 年, K-Means 一词首次由 MacQueen 提出, K-Means 算法作为一种聚类算法, 因为其简单高效, 在静态数据挖掘分析、图像分割等领域得到了广泛应用。K-Means 算法要求预先知道数据被分为多少个 Clusters, 这个与 Mean-Shift 有所不同。

1. 算法解释

K-Means 算法是将集合数据 D 分成为 k 个不同的 Cluster (分类), 每个 Cluster 都有自己的质心点。K-Means 算法的具体步骤如下:

- 1) 假设数据集 D 中包含 n 个数据点 ($D_1 \cdots D_n$)。
- 2) 根据输入的 Cluster 数目 K , 随机初始化 K 个中心点。
- 3) 对数据集 D 中的每个数据点, 计算它到每个 Cluster 中心的距离, 与哪个 Cluster 中心距离最小, 则该点属于哪个 Cluster。
- 4) 对每个 Cluster 中所有数据点计算平均值, 作为新的 Cluster 中心点, 计算新中心与原来中心点差值。
- 5) 重复第 3 ~ 4 步直到收敛 (直到差值变化很小或不变)。

第三步中的距离计算, 最常见是使用欧几里得距离, 此外还可以使用曼哈顿距离或象棋格距离。介绍了 K-Means 算法之后, 下面就是如何将 K-Means 算法运用到图像分割中。一幅图像本质上是由有限个数的像素点组成的, 每个像素点都有 RGB 值。假设全部的像素数据点有 K 个 Cluster, 计算每个像素点到每个 Cluster 中心点的 RGB 空间距离, 与其中一个 Cluster 中心点距离最近的像素点则属于该 Cluster, 对全部的像素点执行同样的操作, 直到收敛就完

成了基于 K-Means 的图像分割。

2. 代码实现

K-Means 图像分割的代码实现大致可以分为如下几步：

1) 根据输入参数 (Cluster 数目)，随机初始化 K-Means 中心点。代码实现如下：

```
//Create random points to use as the cluster center
Random random = new Random();
for (int i = 0; i < numOfCluster; i++)
{
    int randomNumber1 = random.nextInt(width);
    int randomNumber2 = random.nextInt(height);
    index = randomNumber2 * width + randomNumber1;
    ClusterCenter cp = new ClusterCenter(randomNumber1, randomNumber2);
    int argb = inPixels[i];
    int r = (argb >> 16) & 0xff;
    int g = (argb >> 8) & 0xff;
    int b = (argb) & 0xff;
    cp.setRGB(new int[]{r, g, b});
    cp.setIndex(i);
    clusterCenterList.add(cp);
}
```

2) 对所有像素点初始化，完成最初的像素 K-Means 分类。代码实现如下：

```
// create all cluster point
for (int row = 0; row < height; ++row)
{
    for (int col = 0; col < width; ++col)
    {
        index = row * width + col;
        int color = inPixels[index];
        PixelPoint pp = new PixelPoint(row, col);
        int r = (color >> 16) & 0xff;
        int g = (color >> 8) & 0xff;
        int b = (color) & 0xff;
        pp.setRGB(new float[]{r, g, b});
        pp.setLabel(-1);
        pointList.add(pp);
    }
}

// initialize the clusters for each point
double[] clusterDisValues = new double[clusterCenterList.size()];
for(int i=0; i<pointList.size(); i++)
{
    for(int j=0; j<clusterCenterList.size(); j++)
    {
        clusterDisValues[j] = calculateEuclideanDistance(clusterCenterList.
            get(j), pointList.get(i));
    }
}
```

```

    }
    pointList.get(i).setLabel(getCloserCluster(clusterDisValues));
}

```

其中计算像素点与 Cluster 中心点距离的代码如下:

```

private double calculateEuclideanDistance(ClusterCenter p, PixelPoint c)
{
    // each pixel
    int pr = (int)p.getRGB()[0];
    int pg = (int)p.getRGB()[1];
    int pb = (int)p.getRGB()[2];
    // cluster center
    int cr = (int)c.getRGB()[0];
    int cg = (int)c.getRGB()[1];
    int cb = (int)c.getRGB()[2];

    return Math.sqrt(Math.pow((pr - cr), 2.0) + Math.pow((pg - cg), 2.0) + Math.
        pow((pb - cb), 2.0));
}

```

3) 计算分类以后每个 Cluster 的像素点平均值,并用此更新中心点像素值。代码实现如下:

```

private double[] reCalculateClusterCenters() {
    // clear the points now
    for(int i=0; i<clusterCenterList.size(); i++)
    {
        clusterCenterList.get(i).setNumOfPixels(0);
    }

    // recalculate the sum and total of points for each cluster
    double[] redSums = new double[3];
    double[] greenSum = new double[3];
    double[] blueSum = new double[3];
    for(int i=0; i<pointList.size(); i++)
    {
        int cIndex = (int)pointList.get(i).getLabel();
        clusterCenterList.get(cIndex).addNumOfPixel();
        int tr = (int)pointList.get(i).getRGB()[0];
        int tg = (int)pointList.get(i).getRGB()[1];
        int tb = (int)pointList.get(i).getRGB()[2];
        redSums[cIndex] += tr;
        greenSum[cIndex] += tg;
        blueSum[cIndex] += tb;
    }

    double[] oldClusterCentersColors = new double[clusterCenterList.size()];
    for(int i=0; i<clusterCenterList.size(); i++)
    {

```

```

double sum = clusterCenterList.get(i).getNumOfPixels();
int cIndex = clusterCenterList.get(i).getIndex();
int red = (int) (greenSum[cIndex]/sum);
int green = (int) (greenSum[cIndex]/sum);
int blue = (int) (blueSum[cIndex]/sum);
System.out.println("red = " + red + " green = " + green + " blue = " + blue);
int clusterColor = (255 << 24) | (red << 16) | (green << 8) | blue;
clusterCenterList.get(i).setRGB(new int[]{red, green, blue});
oldClusterCentersColors[i] = clusterColor;
}

return oldClusterCentersColors;
}

```

4) 比较两次得到的 Cluster 中心点像素平均值是否相等, 相等则停止, 否则继续步骤 3。

代码实现如下:

```

while(true)
{
    stepClusters();
    double[] newClusterCenterColors = reCalculateClusterCenters();
    if(isStop(oldClusterCenterColors, newClusterCenterColors))
    {
        break;
    }
    else
    {
        oldClusterCenterColors = newClusterCenterColors;
    }
}

```

其中, stepClusters 方法实现的代码如下:

```

private void stepClusters()
{
    // initialize the clusters for each point
    double[] clusterDisValues = new double[clusterCenterList.size()];
    for(int i=0; i<pointList.size(); i++)
    {
        for(int j=0; j<clusterCenterList.size(); j++)
        {
            clusterDisValues[j] = calculateEuclideanDistance(clusterCenterList.get(j), pointList.get(i));
        }
        pointList.get(i).setTable(getCloserCluster(clusterDisValues));
    }
}

```

5) 根据得到的 Cluster 分类, 对每个像素点赋值, 输出结果。代码实现如下:

```
//update the result image
```

```

dest = createCompatibleDestImage(src, null );
index = 0;
int[] outPixels = new int[width*height];
for (int j = 0; j < pointList.size(); j++)
{
    for (int i = 0; i < clusterCenterList.size(); i++)
    {
        PixelPoint p = this.pointList.get(j);
        if (clusterCenterList.get(i).getIndex() == p.getLabel())
        {
            int row = p.getRow(); // row
            int col = p.getCol(); // column
            index = row * width + col;
            int[] rgb = clusterCenterList.get(i).getRGB();
            outPixels[index] = (0xff << 24) | (rgb[0] << 16) | (rgb[1] << 8)
                | rgb[2];
        }
    }
}

```

完整的基于 K-Means 图像分割算法的源代码参见源文件中 12-3 的 KMeansAlgo.java, 其中用到的两个数据结构类 ClusterCenter 和 PixelPoint 同样可以在源文件中相关部分找到。关于 K-Means 实现图像分割的基本原理与实现到这里就介绍完了, 代码实现从易于读者理解的角度出发, 没有过多性能上的考量, 感兴趣的读者可以进一步优化代码实现。

12.4 基于 Fuzzy C-Means 的图像分割

Fuzzy C-Means 算法是一种常见的数据聚合算法, 可对数据完成两个或两个以上的 Cluster 分割, 最常见的应用领域是在数据分析与挖掘方面。该算法由 Dunn 在 1973 年提出, 在 1981 被 Bezdek 改进以后经常用在模式识别领域中。与 K-Means 算法类似, 该算法也需要预先输入 Cluster 的数目。

1. 算法解释

Fuzzy C-means 算法主要用于比较 RGB 空间的每个像素值与 Cluster 中的每个中心点值, 最终给每个像素指派一个值 (0 ~ 1 之间), 说明该像素更接近于哪里的中心点, 对任意一个像素点, 其对所有 Cluster 中心点的可能性之和为 1。简单的举例: 假设图像中有三个聚类 Cluster1、Cluster2、Cluster3, 像素 A 对应的三个聚类的值分别为 a_1 、 a_2 、 a_3 , 根据模糊规则可知, $a_1 + a_2 + a_3 = 1$ 。更进一步, 如果 a_1 最大, 则该像素比较接近于 Cluster1。计算总的对象值 J 的公式如下:

$$J_m = \sum_{i=1}^N \sum_{j=1}^C u_{ij}^m \|x_i - c_j\|^2, \text{ 其中 } 1 \leq m < \infty$$

当 J 的前后两次差值小于指定精度或不变时, 停止继续分割, 输出分割以后的图像。其中

$\|x_i - c_j\|$ 是指第 i 个像素与第 j 个 Cluster 中心之间的欧几里得距离, u_{ij}^m 是指第 i 个像素与第 j 个 Cluster 之间的模糊值, m 是输入参数。计算模糊值 u_{ij}^m 的公式为:

$$u_{ij}^m = \frac{1}{\sum_{l=1}^c \left(\frac{\|x_i - c_l\|}{\|x_i - c_j\|} \right)^{\frac{2}{m-1}}}$$

其中 c_k 的计算公式为:

$$c_k = \frac{\sum_i u_{ij}^m \cdot x_i}{\sum_i u_{ij}^m}$$

当 J_m 值趋向于很小, 而且两次差值小于输入参数 $\varepsilon (0 < \varepsilon < 1)$ 时, 循环收敛, 得到最终输出结果。根据上述基本原理, 基于 Fuzzy C-Means 实现数据 Cluster 分割的步骤如下:

- 1) 随机初始化每个 Cluster 的中心值。
- 2) 初始化每个数据点到每个 Cluster 中心值的距离, 计算 membership 值。
- 3) 计算每个像素点与每个 Cluster 中心的 Fuzzy 值, 完成 Fuzzy C-Means 的一个循环。
- 4) 比较前后两次 J_k 的差值, 如果小于输入的精度值, 则退出循环, 输出结果, 否则继续。

2. 代码实现

本质上讲, 图像可以看成是由有限多个的像素组成的, 因为每个像素又都有颜色值, 因此又可以将图像看成是数据集合, 这样就适用 Fuzzy C-Means 算法了, 自然也就有了实现基于 Fuzzy C-Means 的图像分割算法。编程实现的步骤大致如下:

- 1) 初始化所有像素点值与随机选取每个 Cluster 的中心点, 初始化每个像素点 $P[i]$ 对应 Cluster 的模糊值 $p[i][k]$ 并计算 cluster index。
- 2) 计算总的对象值 J 。
- 3) 计算每个 Cluster 的颜色值, 产生新的 Cluster 中心像素值。
- 4) 计算每个像素对应每个 Cluster 的模糊值 (membership), 更新每个像素的 Cluster Index。
- 5) 再次计算对象值 J , 并与第二步的对象值相减, 如果差值小于指定的精度值或达到最大循环数, 停止计算输出结果图像, 否则继续第 2 ~ 4 步。

要让该算法正确运行, 还需要用户输入如下三个参数。

- ❑ numOfCluster: 图像分割数目。
- ❑ maxIteration: Fuzzy C Means 算法最大迭代数目。
- ❑ accuracy: 循环停止条件, 即精度值。

初始化每个像素点与随机初始化 cluster 中心点的代码如下:

```
// initialization the pixel data
```

```

points = new ArrayList<PixelPoint>();
for (int row = 0; row < src.getHeight(); ++row)
{
    for (int col = 0; col < src.getWidth(); ++col)
    {
        index = row * width + col;
        int color = inPixels[index];
        PixelPoint pp = new PixelPoint(row, col);
        int r = (color >> 16) & 0xff;
        int g = (color >> 8) & 0xff;
        int b = (color) & 0xff;
        pp.setRGB(new float[]{r, g, b});
        points.add(pp);
    }
}

//Create random points to use as the cluster centroids
Random random = new Random();
clusters = new ArrayList<FCClusterCenter>();
for (int i = 0; i < numOfCluster; i++)
{
    int randomNumber1 = random.nextInt(width);
    int randomNumber2 = random.nextInt(height);
    index = randomNumber2 * width + randomNumber1;
    FClusterCenter fccc = new FClusterCenter(randomNumber2, randomNumber1);
    fccc.setOriginalPvalue(inPixels[index]);
    fccc.setPvalue(inPixels[index]);
    clusters.add(fccc);
}

```

初始化计算每个像素点对应每个 cluster 中心点距离的代码如下:

```

// Iterate through all points to create initial U matrix
double diff;
fuzzyForPixels = new double[this.points.size()][this.clusters.size()];
for (int i = 0; i < this.points.size(); i++)
{
    PixelPoint p = points.get(i);
    for (int j = 0; j < this.clusters.size(); j++)
    {
        FClusterCenter c = this.clusters.get(j);
        diff = Math.sqrt(Math.pow(calculateEuclideanDistance(p, c), 2.0));
        fuzzyForPixels[i][j] = (diff == 0) ? Eps : diff;
    }
}

```

其中 calculateEuclideanDistance() 方法用于计算两个像素点之间的欧几里得距离。实现每个像素点到每个 cluster 中心 membership 的可能性系数计算的代码如下:

```

private void recalculateClusterMembershipValues()
{

```



```

for (int i = 0; i < this.points.size(); i++)
{
    double max = 0.0;
    double min = 0.0;
    double sum = 0.0;
    double newmax = 0;
    PixelPoint p = this.points.get(i);
    //Normalize the entries
    for (int j = 0; j < this.clusters.size(); j++)
    {
        max = fuzzyForPixels[i][j] > max ? fuzzyForPixels[i][j] : max;
        min = fuzzyForPixels[i][j] < min ? fuzzyForPixels[i][j] : min;
    }
    //Sets the values to the normalized values between 0 and 1
    for (int j = 0; j < this.clusters.size(); j++)
    {
        fuzzyForPixels[i][j] = (fuzzyForPixels[i][j] - min) / (max - min);
        sum += fuzzyForPixels[i][j];
    }
    //Makes it so that the sum of all values is 1
    for (int j = 0; j < this.clusters.size(); j++)
    {
        fuzzyForPixels[i][j] = fuzzyForPixels[i][j] / sum;
        if (Double.isNaN(fuzzyForPixels[i][j]))
        {
            fuzzyForPixels[i][j] = 0.0;
        }
        newmax = fuzzyForPixels[i][j] > newmax ? fuzzyForPixels[i][j] :
            newmax;
    }
    // ClusterIndex is used to store the strongest membership value to a cluster,
    used for defuzzification
    p.setPossible(newmax);
}
}

```

实现 Fuzzy C-Means 迭代计算的代码如下：

```

int k = 0;
double oldJm = calculateObjectiveFunction();
do
{
    k++;
    calculateClusterCentroids();
    stepFuzzy();
    double Jnew = calculateObjectiveFunction();
    System.out.println("Run method accuracy of delta value = " + Math.abs(oldJm -
        Jnew));
    if (Math.abs(oldJm - Jnew) < accuracy) break;
    oldJm = Jnew;
}

```

```

}
while (maxIteration > k);

```

其中, `stepFuzzy()` 的实现代码如下:

```

public void stepFuzzy()
{
    for (int c = 0; c < this.clusters.size(); c++)
    {
        for (int h = 0; h < this.points.size(); h++)
        {
            double top;
            top = calculateEuclideanDistance(this.points.get(h), this.clusters.
                get(c));
            if (top < 1.0) top = Eps;
            // sumTerms is the sum of distances from this data point to all
            // clusters.
            double sumTerms = 0.0;
            for (int ck = 0; ck < this.clusters.size(); ck++)
            {
                sumTerms += top / calculateEuclideanDistance(this.points.
                    get(h), this.clusters.get(ck));
            }
            // Then the membership value can be calculated as...
            fuzzyForPixels[h][c] = (double)(1.0 / Math.pow(sumTerms, (2 /
                (this.fuzzy - 1))));
        }
        this.recalculateClusterMembershipValues();
    }
}

```

计算总对象值 J 的代码实现如下:

```

public double calculateObjectiveFunction()
{
    double Jk = 0.0;
    for (int i = 0; i < this.points.size(); i++)
    {
        for (int j = 0; j < this.clusters.size(); j++)
        {
            Jk += Math.pow(fuzzyForPixels[i][j], this.fuzzy) * Math.pow(this.ca
                lculateEuclideanDistance(points.get(i), clusters.get(j)), 2);
        }
    }
    return Jk;
}

```

更新每个 Cluster 中心像素平均值的代码如下：

```
public void calculateClusterCentroids()
{
    for (int j = 0; j < this.clusters.size(); j++)
    {
        FuzzyClusterCenter clusterCentroid = this.clusters.get(j);

        double l = 0.0;
        clusterCentroid.setRedSum(0);
        clusterCentroid.setBlueSum(0);
        clusterCentroid.setGreenSum(0);
        clusterCentroid.setMembershipSum(0);
        double redSum = 0;
        double greenSum = 0;
        double blueSum = 0;
        double membershipSum = 0;
        // double pixelCount = 1;

        for (int i = 0; i < this.points.size(); i++)
        {
            PixelPoint p = this.points.get(i);
            l = Math.pow(fuzzyForPixels[i][j], this.fuzzy);
            int tr = (int)p.getRGB()[0];
            int tg = (int)p.getRGB()[1];
            int tb = (int)p.getRGB()[2];
            redSum += l * tr;
            greenSum += l * tg;
            blueSum += l * tb;
            membershipSum += l;
        }

        int clusterColor = (255 << 24) | ((int)(redSum / membershipSum) <<
            16) | ((int)(greenSum / membershipSum) << 8) | (int)(blueSum /
            membershipSum);
        clusterCentroid.setPvalue(clusterColor);
    }
}
```

根据计算返回值，显示最终分割以后图像的代码如下：

```
//update the original image
dest = createCompatibleDestImage(src, null );
index = 0;
int[] outPixels = new int[width*height];
for (int j = 0; j < this.points.size(); j++)
{
    for (int i = 0; i < this.clusters.size(); i++)
    {
        PixelPoint p = this.points.get(j);
```

```

        if (fuzzyForPixels[j][i] == p.getPossible())
        {
            int row = (int)p.getRow(); // row
            int col = (int)p.getCol(); // column
            index = row * width + col;
            outPixels[index] = this.clusters.get(i).getPvalue();
        }
    }

    // fill the pixel data
    setRGB( dest, 0, 0, width, height, outPixels );
    return dest;

```

从上面的代码也可以看出, 最终每个像素被分配到与之最相似的 Cluster 中, K-Means 则每次都会不断地对每个像素进行分割, 而 Fuzzy C-Means 不断地迭代计算可能性, 最后分割。所以有人把基于 K-Means 实现的图像分割叫做硬分割, 而把基于 Fuzzy C-Means 或类似算法的图像分割叫做软分割。

完整的 Fuzzy C-Means 图像分割源代码可以参见源文件中 12-4 的 FuzzyCMeansAlgo.java, 其相关数据结构类 FCCLusterCenter.java 同样可以在那里得到。读者可以尝试运作与修改源代码, 让其更加有效率地执行。

12.5 基于分水岭的图像分割

图像形态学是数字图像处理一个分支学科, 图像分水岭算法是图像形态学中重要的算法之一, 最常见的是通过 Luc Vincent 和 Pierre Soille 提出的漫水浸泡算法来实现。对彩色图像进行分割时, 首先要将彩色图像变为灰度级别的梯度图像, 然后使用分水岭变换实现图像分割, 这个直接输出的结果往往被过度分割, 所以还需要后续处理, 通过直方图相似度实现合并, 才能得到最终的分割以后的图像。

1. 基本思路

常见的彩色图像分水岭分割方法是先将图像转换为灰度图像, 然后进行分割处理, 最常见的处理有如下几种方式:

- ❑ 基于距离变化实现灰度级别调整, 为分水岭变换分割做好准备。
- ❑ 基于梯度计算实现灰度级别调整, 为分水岭变换分割做好准备。
- ❑ 基于标记控制的分水岭变换, 要实现自动标记寻找。

上述三种途径均可以实现图像分水岭分割, 基于距离变换或梯度的方法容易导致图像过度分割, 需要后续进行合并, 基于标记控制的分水岭算法可很好地避免这一问题。但是基于标记控制的分水岭分割会涉及更多的图像形态学知识, 所以基于本书前面章节的知识基础, 这里所采用的方法是基于梯度图像来完成图像分水岭分割, 梯度计算采用 Sobel 算子完成, 使

用分水岭算法对图像分割以后，使用第 6 章中提到直方图相似度比较方法实现对过度分割图像块的合并。基于上述基本思想，这里实现图像分水岭分割的步骤大致如下：

- 1) 将输入的彩色图像转换为灰度图像。
- 2) 使用 Sobel 算子对灰度图像实现梯度计算。
- 3) 使用梯度图像完成图像分水岭变换。
- 4) 对分水岭变换以后的图像进行组件标记。
- 5) 基于巴氏距离，使用直方图相似度比较方法实现分割组件合并。
- 6) 根据组件内像素平均值之间的距离，合并删除较小分割组件。
- 7) 输出结果。

上述各个步骤对以前章节介绍的知识都有涉及，相关知识点此处不再赘述，读者如有疑问可以查阅相关章节。这里要强调的是图像分水岭分割的方法有很多，虽然实践上千差万别，但是其基本思想都基于图像分水岭变换算法，同时要解决图像过度分割的问题。

2. 代码实现

基于上述思路，实现代码大致可以分为如下几个步骤：

- 1) 对输入彩色图像完成灰度化，实现代码片段如下：

```
// 图像灰度化
int[] inPixels = new int[width*height];
int[] outPixels = new int[width*height];
getRGB( src, 0, 0, width, height, inPixels );
int index = 0;
for(int row=0; row<height; row++) {
    int ta = 0, tr = 0, tg = 0, tb = 0;
    for(int col=0; col<width; col++) {
        index = row * width + col;
        ta = (inPixels[index] >> 24) & 0xff;
        tr = (inPixels[index] >> 16) & 0xff;
        tg = (inPixels[index] >> 8) & 0xff;
        tb = inPixels[index] & 0xff;
        int gray= (int)(0.299 *tr + 0.587*tg + 0.114*tb);
        outPixels[index] = gray;
    }
}
```

- 2) 基于灰度图像实现梯度计算，实现代码片段如下：

```
// 梯度计算
int[] gradientResult = gradient(outPixels, width, height);
setRGB( dest, 0, 0, width, height, gradientResult );
```

- 3) 基于梯度图像实现分水岭变换，实现代码片段如下：

```
// 分水岭变换
WatershedTransform wt = new WatershedTransform();
dest = wt.filter(dest, null);
```

4) 基于分水岭变换结果, 完成组件标记, 实现代码片段如下:

```
// 区域标记
getRGB( dest, 0, 0, width, height, outPixels );
List<PixelPoint> pixelList = new ArrayList<PixelPoint>();

// 初始化每个像素节点状态
for(int row=0; row<height; row++) {
    for(int col=0; col<width; col++) {
        index = row * width + col;
        PixelPoint p = new PixelPoint(row, col, (outPixels[index] >> 16) & 0xff);
        pixelList.add(p);
    }
}

// 添加每个像素节点的四邻域像素
for(int row=0; row<height; row++) {
    for(int col=0; col<width; col++) {
        index = row * width + col;
        PixelPoint p = pixelList.get(index);

        // add four neighbors for each pixel
        if((row - 1) >= 0) {
            index = (row-1) * width + col;
            p.addNeighbour(pixelList.get(index));
        }
        if((row + 1) < height) {
            index = (row+1) * width + col;
            p.addNeighbour(pixelList.get(index));
        }
        if((col - 1) >= 0) {
            index = row * width + col-1;
            p.addNeighbour(pixelList.get(index));
        }
        if((col+1) < width) {
            index = row * width + col+1;
            p.addNeighbour(pixelList.get(index));
        }
    }
}

// 深度优先搜索算法, 连通组件标记
DFSAlgorithm dfs = new DFSAlgorithm(pixelList);
dfs.process();
```

5) 基于组件标记结果, 完成巴氏直方图相似度合并, 其中要求相似度大于 0.8。实现代码片段如下:


```

// 直方图相似度合并
Integer[] labelKeys = labelPixelMap.keySet().toArray(new Integer[0]);
double[][] allBins = new double[labelKeys.length][4*4*4];
for(int k=0; k<labelKeys.length; k++)
{
    ArrayList<PixelPoint> labeledPixels = labelPixelMap.get(labelKeys[k]);
    allBins[k] = calculateHistogram(labeledPixels, width, inPixels);
}
Arrays.fill(outPixels, -1);
for(int k=0; k<labelKeys.length; k++)
{
    if(!labelPixelMap.containsKey(labelKeys[k])) continue;
    ArrayList<PixelPoint> labeledPixels = labelPixelMap.get(labelKeys[k]);
    double[] srcBins = allBins[k];
    for(int i=0; i<labelKeys.length; i++)
    {
        if(k == i) continue;
        if(!labelPixelMap.containsKey(labelKeys[i])) continue;
        double[] destBins = allBins[i];
        if(calculateBhattacharyya(srcBins, destBins)>0.8)
        {
            int megerLabel = labeledPixels.get(0).getLabel();
            for(PixelPoint mp : labelPixelMap.get(labelKeys[i]))
            {
                index = mp.getY() * width + mp.getX();
                outPixels[index] = megerLabel;
            }
            labelPixelMap.remove(labelKeys[i]);
        }
    }
    for(PixelPoint pp : labeledPixels)
    {
        index = pp.getY() * width + pp.getX();
        outPixels[index] = pp.getLabel();
    }
}

```

6) 删除较小组件块，假设组件中包含的像素数目小 3000，实现的代码片段如下：

```

for(int i=0; i<labelKeys.length; i++)
{
    double minDis = Double.MAX_VALUE;
    int size = ccounts[i];
    if(size > 3000) continue;
    int tag = -1;
    int foundIndex = -1;
    for(int j=0; j<labelKeys.length; j++)
    {
        if(j == i) continue;
        if(ccounts[j] < 3000) continue;
        double dis = calculateEuclideanDistance(cmeans[j], cmeans[i]);
        if(dis < minDis)

```

```

        {
            minDis = dis;
            foundIndex = j;
        }
    }
    if(foundIndex > 0)
    {
        for(int f=0; f<outPixels.length; f++)
        {
            if(outPixels[f] == labelKeys[i]){
                outPixels[f] = labelKeys[foundIndex];
            }
        }
    }
}
}

```

7) 输出分割图像的代码片段如下:

```

colorMap.clear();
for(int i=0; i<outPixels.length; i++)
{
    Color c = getColor(outPixels[i], colorMap);
    outPixels[i] = (0xff << 24) | (c.getRed() << 16) | (c.getGreen() << 8) |
        c.getBlue();
}

```

完整的程序源代码参见源文件中 12-5 的 WatershedSegmentationAlgo.java, 其中组件标记中用的深度搜索 DFSAlgorithm.java 参见源文件中第 10 章的相关部分, WatershedTransform.java 参见源文件中第 11 章相关部分即可。

图像分水岭变换没有固定的步骤, 这里给出的彩色图像分水岭分割是笔者自己总结与摸索处理的。分水岭分割最主要的缺点是容易导致图像过度分割, 解决这个问题的方法有很多, 希望读者可以自己尝试, 在实践中磨砺自己的技能。

12.6 小结

本章主要介绍了几种常见的图像分割方法, 在介绍这些方法时, 尽量做到少讲空话, 少用那些生涩的数学公式, 而多用语言描述与简单举例来帮助大家理解相关知识, 通过详细的代码实现步骤与完整源代码演示了每一种介绍的图像分割方法。常见的图像分割方法还有基于图与拓扑的最大流与最小割方法, 基于高斯混合模型的分割方法等, 希望读者能够进一步阅读相关资料, 在理解与掌握本章内容的基础上继续学习。特别强调的是源代码是本书的一部分, 请读者阅读、运行、修改源代码, 完成自己版本的各种图像分割方法。

另外, 本章中涉及的数学知识主要是向量之间的距离计算, 并没有介绍特别复杂的数学知识与公式推导, 希望读者自己阅读相关图像分割方法的原理数学证明与推导, 这对掌握本章知识大有帮助。

Chapter 13 第 13 章

图像特征的提取与检测

上一章介绍了几种常见的图像分割方法，加上对前面所学知识的理解与运用。本章将继续基于已经学习的章节知识深入了解图像特征提取这一图像处理领域的重要分支，当然本章内容有限，不可能深入到图像特征提取的方方面面，这里选取介绍的内容都是图像特征提取中比较常见而且重要的基础知识。

图像特征可能包括图像中对象的主要特性，比如边缘、角点、纹理、颜色等。图像特征提取在模式匹配、图像识别、对象检测等方面对工业检测、机器人视觉等领域有着重要意义。常见的图像特征提取有直线检测、边缘检测、角点检测、纹理提取、图像多尺度金字塔特征等，这些内容本章均有涉及，最后还将详细剖析经典 SIFT 特征提取算法。

13.1 颜色特征提取

因为图像像素值数组是一系列的数值，所以从本质上来说，可以通过一些简单的数学统计学知识来实现对图像整体颜色特征的表述。最常见的颜色特征就是基于 RGB 色彩空间每个通道计算 Moments。

一阶 Moments 表示均值 (means) 计算公式为：

$$\mu = \sum_{j=0}^N \frac{1}{N} P_{ij}$$

其中 P_{ij} 表示第 i 个颜色通道在像素位置 j 上的值、 N 表示总的像素个数。

二阶 Moments 是标准方差 (Standard Deviation)，是分布差的平方根，计算公式可以表示为：

$$\sigma = \sqrt{\left(\frac{1}{N} \sum_{j=1}^N (P_{ij} - \mu_i)^2\right)}$$

三阶 Moments 表示偏斜度 (Skewness), 它给出了像素值分布的不对称度量, 计算公式可以表示为:

$$Skewness = \sqrt[3]{\left(\frac{1}{N} \sum_{j=1}^N (P_{ij} - \mu_i)^3\right)}$$

若对 RGB 像素值的每个通道都计算三个值, 就可以得到 RGB 色彩空间的 9 个值; 转换到 HSV 色彩空间, 对 H、S、V 通道同样计算这三个值, 就可以得到 18 个颜色特征值。编程实现的步骤大致如下:

- 1) 获取输入图像的像素数据数组。
- 2) 分别在 RGB 色彩空间与 HSV 色彩空间计算均值、方差、偏斜度。
- 3) 得到输出的 18 个颜色特征值。

实现代码如下:

```
package com.book.chapter.thirteen;

import java.awt.Color;
import java.awt.image.BufferedImage;

import com.book.chapter.four.AbstractBufferedImageOp;

public class ColorFeatureExtractor extends AbstractBufferedImageOp {

    @Override
    public BufferedImage filter(BufferedImage src, BufferedImage dest) {
        int width = src.getWidth();
        int height = src.getHeight();
        if (dest == null)
            dest = createCompatibleDestImage(src, null);
        int[] inPixels = new int[width*height];
        src.getRGB(0, 0, width, height, inPixels, 0, width);
        int index = 0;

        // 提取像素
        float[][] R=new float[height][width];
        float[][] G=new float[height][width];
        float[][] B=new float[height][width];
        float[][] H=new float[height][width];
        float[][] S=new float[height][width];
        float[][] V=new float[height][width];
        float hsv[]=new float[3];
        for (int row=0;row<height;row++){
            for (int col=0;col<width;col++){
                index = row * width + col;
                int argb = inPixels[index];
                int r = (argb >> 16) & 0xff;
```

```

        int g = (argb >> 8) & 0xff;
        int b = (argb) & 0xff;
        R[row][col]=r;
        G[row][col]=g;
        B[row][col]=b;
        hsv=Color.RGBtoHSB(r, g, b, hsv);
        H[row][col]=hsv[0];
        S[row][col]=hsv[1];
        V[row][col]=hsv[2];
    }

    // 提取颜色特征值
    double[] redMSS = meanStdSkew(R, height, width);
    double[] greenMSS = meanStdSkew(G, height, width);
    double[] blueMSS = meanStdSkew(B, height, width);
    double[] hMSS = meanStdSkew(H, height, width);
    double[] sMSS = meanStdSkew(S, height, width);
    double[] vMSS = meanStdSkew(V, height, width);

    // 返回结果
    System.out.println("h = " + hMSS[0]);
    System.out.println("s = " + sMSS[0]);
    System.out.println("v = " + vMSS[0]);
    for (int row=0;row<height;row++){
        for (int col=0;col<width;col++){
            index = row * width + col;
            inPixels[index] = (0xff << 24) | (((int)redMSS[0]) << 16)
                | (((int)greenMSS[0]) << 8) | ((int)blueMSS[0]);
        }
        setRGB(dest, 0, 0, width, height, inPixels);
    }
    return dest;
}

public static double[] meanStdSkew( float[][] data, int height, int width )
{
    double mean = 0;
    double[] out=new double[3];

    for (int row=0;row<height;row++){
        for (int col=0;col<width;col++){
            mean += data[row][col];
        }
    }
    mean /= (height*width);
    out[0]=mean;
    double sum = 0;
    for (int row=0;row<height;row++){
        for (int col=0;col<width;col++){
            final double v = data[row][col] - mean;
            sum += v * v;
        }
    }
}

```

```

    }
    }
    out[1]=Math.sqrt( sum / ( height*width - 1 ) );
    sum = 0;
    for (int row=0;row<height;row++){
        for (int col=0;col<width;col++){
            final double v = (data[row][col] - mean)/out[1];
            sum += v * v * v;
        }
    }
    out[2]=Math.pow(1+sum/(height*width-1),1./3);
    return out;
}
}

```

颜色特征计算的应用之一是将其作为基本属性建立图像基于颜色的索引,要运行上述代码,在 ImagePanel.java 的 process() 方法中添加如下调用代码即可(第 3 章有介绍):

```

ColorFeatureExtractor filter = new ColorFeatureExtractor();
destImage = filter.filter(sourceImage, null);

```

本章余下内容的代码调用都与该处类似,后续将不再赘述,如果使用各种特征提取代码,读者只需要参见这里的代码,替换其中的类即可。

13.2 纹理提取

纹理是图像中常见的特征之一,常见的三种纹理包括随机块、方向块、连续块,如图 13-1 所示。

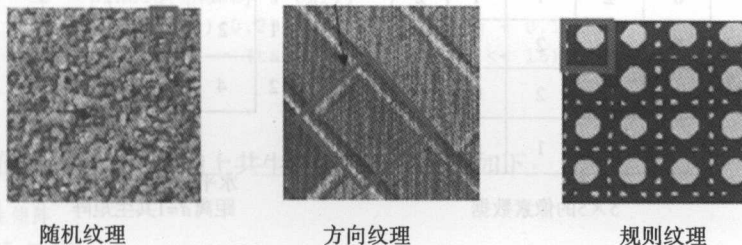


图 13-1 纹理特征

针对不同类型纹理与特征,纹理提取的方法也有很多,最常见两种统计方法是基于直方图与共生矩阵提取纹理特征,另外一种是基于图像空间域 Gabor 滤波提取。较复杂的有基于图像频率域傅里叶变换与图像多尺度小波实现纹理特征提取,这两种方法本书不会讨论,感兴趣的读者可以自己研究。本节主要介绍基于共生矩阵实现纹理提取的方法。

1. 灰度共生矩阵 (co-occurrence matrices) 概念

由于图像纹理是灰度分布在位置上反复出现而形成的，因此在图像空间域里，相隔某种距离的两个像素之间一定存在着某种联系，这种联系即图像像素的空间相关性，灰度共生矩阵就是研究图像灰度空间相关性来进行纹理特征描述的一种常见方法。

灰度共生矩阵是基于灰度图像得到的，基于统计灰度像素点 i 的灰度值在 0° 、 45° 、 90° 、 135° 方向与指定距离的像素点 j 的灰度值同时出现次数而生成的矩阵，可以表示为 $P(i, j)$ ，其中 i, j 表示图像灰度级别，四个角度分量可以表示为 P_0 、 P_{45} 、 P_{90} 、 P_{135} ，如图 13-2 所示。

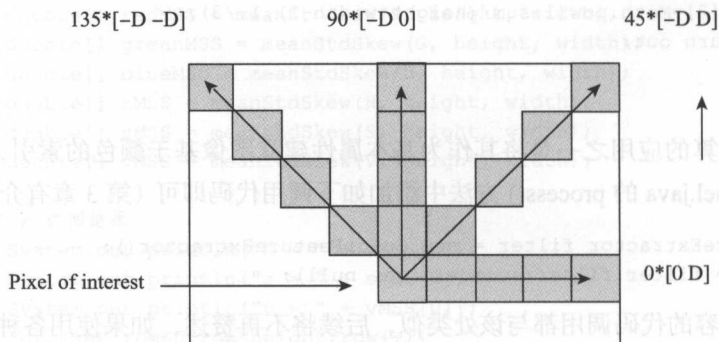


图 13-2 共轭矩阵

其中 D 表示距离。假设有灰度级别为 0、1、2 的 5×5 图像像素数据，对每个像素周围水平与垂直方向进行共生矩阵统计，得到的结果如图 13-3 所示。

2	1	2	0	1
0	2	1	1	2
0	1	2	2	0
1	2	2	0	1
2	0	1	0	1

5×5的像素数据

	0	1	2
0	2	0	4
1	2	2	5
2	4	8	3

水平与垂直方向，
距离 $d=1$ 共生矩阵

图 13-3 像素 5×5 共轭矩阵计算结果

根据统计获得的灰度共生矩阵可以计算纹理下列相关属性。

□ 对比度 (Contrast): 计算像素与周围像素对整个图像灰度的度量，0 表示为常量图像，计算公式为：

$$\text{contrast} = \sum_{i,j} |i - j|^2 p(i, j)$$

- 相关性 (Correlation): 计算像素与周围像素之间相关性高低, 取值范围为 $[-1, 1]$, 计算公式为:

$$\text{correlation} = \sum_{i,j} \frac{(i - \mu_i)(j - \mu_j)p(i,j)}{\sigma_i \sigma_j}$$

- 能量 (Energy): 计算矩阵中每个元素的平方之和, 1 表示为常量图像。计算公式为:

$$\text{energy} = \sum_{i,j} p(i,j)^2$$

2. 编程实现

根据上述介绍, 编程实现图像灰度共生矩阵统计的大致步骤如下:

- 1) 将输入图像转换为灰度图像。
- 2) 读取每个像素值, 根据灰度值生成共生矩阵。
- 3) 计算对比度、相关性、能量值。
- 4) 输出结果图像。

程序在完成时还需要考虑距离 D 参数的大小, 这里默认距离 $D = 1$ 。根据步骤描述, 实现图像转为灰度图像的代码如下:

```
// 图像灰度化
int[] pixels = new int[width*height];
getRGB( src, 0, 0, width, height, pixels );
int index = 0;
for(int row=0; row<height; row++) {
    int ta = 0, tr = 0, tg = 0, tb = 0;
    for(int col=0; col<width; col++) {
        index = row * width + col;
        ta = (pixels[index] >> 24) & 0xff;
        tr = (pixels[index] >> 16) & 0xff;
        tg = (pixels[index] >> 8) & 0xff;
        tb = pixels[index] & 0xff;
        int gray= (int)(0.299 *tr + 0.587*tg + 0.114*tb);
        pixels[index] = (ta << 24) | (gray << 16) | (gray << 8) | gray;
    }
}
```

根据像素值, 建立四个方向上共生矩阵的代码实现如下:

```
// 计算共生矩阵
int offset = 0;
double totalPixels = 0;
double[][] coMatrix = new double[256][256];
for(int row=0; row<height; row++) {
    for(int col=0; col<width; col++) {
        index = row * width + col; // i
        int igray = (pixels[index] >> 16) & 0xff;
        if(degrees == ZERO_DEGREES)
        {
            offset = col + distance;
```

```

if(offset >= width)
{
    continue;
}
index = row * width + offset;
int jgray = (pixels[index] >> 16) & 0xff;
coMatrix[igray][jgray] += 1;
coMatrix[jgray][igray] += 1;
totalPixels += 2;
}
else if(degrees == ANGLE_45_DEGREES)
{
    offset = col + distance;
    if(offset >= width)
    {
        continue;
    }
    if((row - distance) < 0)
    {
        continue;
    }
    index = (row-distance) * width + offset;
    int jgray = (pixels[index] >> 16) & 0xff;
    coMatrix[igray][jgray] += 1;
    coMatrix[jgray][igray] += 1;
    totalPixels += 2;
}
else if(degrees == ANGLE_90_DEGREES)
{
    offset = row - distance;
    if(offset < 0)
    {
        continue;
    }
    index = (offset) * width + col;
    int jgray = (pixels[index] >> 16) & 0xff;
    coMatrix[igray][jgray] += 1;
    coMatrix[jgray][igray] += 1;
    totalPixels += 2;
}
else if(degrees == ANGLE_135_DEGREES)
{
    offset = col - distance;
    if(offset < 0)
    {
        continue;
    }
    if((row - distance) < 0)
    {
        continue;
    }
    offset = col + distance;
}

```

```

index = (row-distance) * width + offset;
int jgray = (pixels[index] >> 16) & 0xff;
coMatrix[igray][jgray] += 1;
coMatrix[jgray][igray] += 1;
totalPixels += 2;
}
}

```

根据灰度共生矩阵数据计算对比度的代码实现如下:

```

// calculates the contrast
double contrast=0.0;
for (int i=0; i<256; i++) {
    for (int j=0; j<256; j++) {
        contrast=contrast+ (i-j)*(i-j)*(coMatrix[i][j]);
    }
}

```

根据灰度共生矩阵数据计算能力的代码实现如下:

```

// calculates the angular second moment - ASM
double asm=0.0;
for (int i=0; i<256; i++) {
    for (int j=0; j<256; j++) {
        asm=asm+ (coMatrix[i][j]*coMatrix[i][j]);
    }
}

```

根据灰度共生矩阵数据计算相关性的代码实现如下:

```

// 计算和
double pi = 0;
double pj = 0;
for (int i=0; i<256; i++){
    for (int j=0; j<256; j++){
        pi=pi+i*coMatrix [i][j];
        pj=pj+j*coMatrix [i][j];
    }
}
// 计算方差
double stdevi = 0;
double stdevj = 0;
for (int i=0; i<256; i++){
    for (int j=0; j<256; j++){
        stdevi=stdevi+(i-pi)*(i-pi)*coMatrix [i][j];
        stdevj=stdevj+(j-pj)*(j-pj)*coMatrix [i][j];
    }
}
// 计算相关性
double correlation = 0;
for (int i=0; i<256; i++) {

```

```
for (int j=0; j<256; j++) {
    correlation=correlation+( (i-pi)*(j-pj)*coMatrix [i][j]/(stdevi*stdevj)) ;
}
}
```

完整的灰度共生矩阵代码实现参见源文件中 13-02 的 CoOccurrenceMatrixFilter.java 即可，运行与测试该代码可以参考以前章节中提到的调用方法。这里需要特别说明一下的是，上述步骤中计算得到的灰度共生矩阵在计算对比度等属性之前需要归一化处理，具体代码参见源代码文件，这里不再赘述。

13.3 直线检测

直线也是图像中几何特征的一种，水平或垂直方向的直线可以使用图 13-4 所示的两个检测算子。

角度为 45° 或 135° 方向的直线则使用图 13-5 中的检测算子。

-1	2	-1
-1	2	-1
-1	2	-1

垂直方向

-1	-1	-1
2	2	2
-1	-1	-1

水平方向

图 13-4 水平与垂直方向算子

-1	-1	2
-1	2	-1
2	-1	-1

45度方向

2	-1	-1
-1	2	-1
-1	-1	2

135度方向

图 13-5 斜 45、135 度方向算子

上述四个算子只能对一些简单图像实现直线检测，对于一些复杂的图像对象，直线很难做到正确检测。在实际应用中，更为人知的直线检测方法是基于霍夫变换实现的，霍夫变换是经典的图像变换方法之一，主要用来从图像中分离具有某种相同特征的几何形状，比如圆、直线等。霍夫变换检测直线的方法与其他方法相比，可以更好地抗噪声干扰。

1. 基本原理

霍夫变换检测直线的基本原理是把每个像素坐标点经过变换都变成对直线特质有贡献的统一度量。一个简单的例子如下：一条直线在图像中是一系列离散点的集合，通过一个直线的离散极坐标公式，可以表达出直线的离散点几何等式为： $r = x\cos(\theta) + y\sin(\theta)$ ，其中角度 θ 指 r 与 X 轴之间的夹角， r 为到直线几何垂直距离。任何在直线上的点， x 、 y 都可以表达，其中 r 、 θ 是常量。该公式的图形表示如图 13-6 所示。

然而在实际的图像处理过程中，一般图像的像素坐标 $P(x, y)$ 是已知的， r 、 θ 则是我们要寻找的变量。如果能根据像素点坐标 $P(x, y)$ 绘制每个 (r, θ) ，那么就将图像从笛卡儿坐标系统转



图 13-6 直线极坐标

换到极坐标霍夫空间系统, 这种从点到曲线的变换称为直线的霍夫变换。在使用霍夫变换算法时, 每个像素坐标点 $P(x, y)$ 都会被转换到 (r, θ) 的曲线点上面, 累加到对应的格子数据点, 当一个波峰出现时, 说明有直线存在。这里假设 θ 的取值范围为 $[0^\circ, 180^\circ]$ 之间, 步长为 1, 当然为了获得更加准确的结果可以将步长变小。

2. 实现

根据上述原理, 实现利用霍夫变换完成直线检测大致需要如下几步:

1) 初始化霍夫变换空间、极坐标空间, 角度为 $0^\circ \sim 180^\circ$, 最大可能半径长度为输入图像宽与高的平方根。

2) 将图像的 2D 空间转换到霍夫空间, 每个像素坐标都要转换到霍夫极坐标的对应强度值。

3) 找出霍夫极坐标空间的最大强度值。

4) 根据最大强度值归一化, 范围为 $0 \sim 255$ 。

5) 根据输入前 accSize 值找出前 accSize 个信号最强的直线。

6) 根据在霍夫空间找到的结果, 在像素空间画出检测得到的直线。

第一步, 初始化霍夫变换空间的代码实现如下:

```
int rmax = (int) Math.sqrt(width * width + height * height);
acc = new int[rmax * 180]; // 0 ~ 180 霍夫坐标空间
int r;
```

第二步, 将图像 2D 空间像素点 $P(x, y)$ 从笛卡儿坐标变换为极坐标的代码实现如下:

```
for (int x = 0; x < width; x++) {
    for (int y = 0; y < height; y++) {
        if ((input[y * width + x] & 0xff) == 255) {
            for (int theta = 0; theta < 180; theta++) {
                r = (int) (x * Math.cos(((theta) * Math.PI) / 180) + y *
                    Math.sin(((theta) * Math.PI) / 180)); // 像素点 p(x, y) 转换
                为对应的强度值
                if ((r > 0) && (r <= rmax))
                    acc[r * 180 + theta] = acc[r * 180 + theta] + 1;
                // 强度值增加1
            }
        }
    }
}
```

第三步, 找出霍夫变换以后极坐标空间的最大强度值的代码实现如下:

```
// 找到极坐标空间的最大强度值
int max = 0;
for (r = 0; r < rmax; r++) {
    for (int theta = 0; theta < 180; theta++) {
```



```

        if (acc[r * 180 + theta] > max) {
            // swap the max value
            max = acc[r * 180 + theta];
        }
    }
}

```

第四步，根据最大与最小强度值，完成归一化的代码实现如下：

```

// normalization all the values, 归一化处理
int value;
for (r = 0; r < rmax; r++) {
    for (int theta = 0; theta < 180; theta++) {
        value = (int) (((double) acc[r * 180 + theta] / (double) max) * 255.0);
        acc[r * 180 + theta] = 0xff000000 | (value << 16 | value << 8 | value);
    }
}

```

第五步，根据输入 N 在霍夫空间中寻找前 N 个最大值的 r 、 θ ，以及强度值，代码如下：

```

// 极坐标中最大的半径值
int rmax = (int) Math.sqrt(width * width + height * height);
results = new int[accSize * 3];
int[] output = new int[width * height];
// 根据输入参数，找到前accSize个极坐标空间中强度最大的直线R值
for (int r = 0; r < rmax; r++) {
    for (int theta = 0; theta < 180; theta++) {
        int value = (acc[r * 180 + theta] & 0xff);

        // if its higher than lowest value add it and then sort
        if (value > results[(accSize - 1) * 3]) {

            // add to bottom of array
            results[(accSize - 1) * 3] = value;
            results[(accSize - 1) * 3 + 1] = r;
            results[(accSize - 1) * 3 + 2] = theta;

            // shift up until its in right place
            int i = (accSize - 2) * 3;
            while ((i >= 0) && (results[i + 3] > results[i])) {
                for (int j = 0; j < 3; j++) {
                    int temp = results[i + j];
                    results[i + j] = results[i + 3 + j];
                    results[i + 3 + j] = temp;
                }
                i = i - 3;
                if (i < 0)
                    break;
            }
        }
    }
}
}

```

第六步, 根据第五步的结果, 在像素空间标出找到前 N 个直线位置, 用红色标记, 代码如下:

```
// 根据极坐标记录的R值, 匹配对应的像素点, 绘制出检测到的直线
System.out.println("Total " + accSize + " matches:");
for (int i = accSize - 1; i >= 0; i--) {
    drawPolarLine(results[i * 3], results[i * 3 + 1], results[i * 3 + 2]);
}
```

方法 drawPolarLine() 的实现代码如下:

```
// 绘制直线方法
private void drawPolarLine(int value, int r, int theta) {
    for (int x = 0; x < width; x++) {
        for (int y = 0; y < height; y++) {
            int temp = (int) (x * Math.cos(((theta) * Math.PI) / 180) + y * Math.
                sin(((theta) * Math.PI) / 180));
            if ((temp - r) == 0) // 匹配对应像素点, 绘制直线
                output[y * width + x] = 0xffff0000; // 结果直线为红色
        }
    }
}
```

完整的霍夫变换直线检测实现源代码参见源文件中 13-03 的 LineHough.java 即可, 其中参数 accSize 表示程序检测之后输出待检测直线的数目。

完整的测试代码 HoughFilter.java 同样位于源文件中的 13-03。这里直线检测是基于二值图像实现的, 同时直线中有段点可能会影响结果, 感兴趣的读者可以自己进一步探索。

13.4 圆检测

上一节介绍了基于霍夫变换实现直线检测的基本原理与方法, 本节介绍基于霍夫变换实现圆检测的基本原理与方法, 基于霍夫变换来检测圆也是霍夫变换的经典应用之一, 在很多开源的图像处理库中都有实现, 本节主要探讨基本原理与方法, 不涉及优化与广义霍夫检测等内容。

1. 基本原理

对于任意 2D 空间上的像素点 $P(x, y)$, 如果属于圆上一点, 则转换为极坐标参数方程如下:

$$\begin{aligned}x &= x_0 + r \cos \theta \\y &= y_0 + r \sin \theta\end{aligned}$$

其中 (x_0, y_0) 表示圆心坐标, θ 值表示旋转角度, r 表示圆的半径, 所以对于任意一个圆, 假设圆中心像素点 $P(x_0, y_0)$ 已知, 圆半径 r 已知, 则旋转 360° 由极坐标方程可以得到该圆每个点上的坐标; 同样如果已经知道图像上像素点 $P(x, y)$ 、圆半径 r 、旋转 360° , 则得到圆心点处的坐标值必定最强。这正是霍夫变换检测圆的数学原理。

2. 实现

根据上述基本原理, 编程实现霍夫变换大致可以分为如下几步:

- 1) 将图像中非背景像素点从 2D 平面空间转换到极坐标空间。
- 2) 在极坐标空间中使用最大最小方法归一化, 值范围为 0 ~ 255 之间。
- 3) 根据输入参数 N , 寻找前 N 个信号最强的圆心。
- 4) 根据输入参数半径大小, 绘制出当前检测结果。
- 5) 返回结果像素数组, 显示为图片。

上述实现霍夫圆检测的步骤中需要输入圆半径参数 r 与待检测圆的数目。其中对于圆半径参数, 还可以改为最小值 R_{\min} 与最大值 R_{\max} 之间的范围来进行检测。分步骤代码详解如下。

第一步, 将图像中非背景像素点从平面空间转换到极坐标空间的代码实现如下:

```
// 对于圆的极坐标变换来说, 我们需要360度的空间梯度叠加值
acc = new int[width * height];
for (int y = 0; y < height; y++) {
    for (int x = 0; x < width; x++) {
        acc[y * width + x] = 0;
    }
}
int x0, y0;
double t;
for (int x = 0; x < width; x++) {
    for (int y = 0; y < height; y++) {
        if ((input[y * width + x] & 0xff) == 255) {
            for (int theta = 0; theta < 360; theta++) {
                t = (theta * 3.14159265) / 180; // 角度值0 ~ 2*PI
                x0 = (int) Math.round(x - r * Math.cos(t));
                y0 = (int) Math.round(y - r * Math.sin(t));
                if (x0 < width && x0 > 0 && y0 < height && y0 > 0) {
                    acc[x0 + (y0 * width)] += 1;
                }
            }
        }
    }
}
```

第二步, 在极坐标空间完成数组归一化的代码实现如下:

```
// Find max acc value
for (int x = 0; x < width; x++) {
    for (int y = 0; y < height; y++) {
        if (acc[x + (y * width)] > max) {
            max = acc[x + (y * width)];
        }
    }
}
```

```

    }
}

// 根据最大值, 实现极坐标空间的灰度值归一化处理
int value;
for (int x = 0; x < width; x++) {
    for (int y = 0; y < height; y++) {
        value = (int) (((double) acc[x + (y * width)] / (double) max) * 255.0);
        acc[x + (y * width)] = 0xff000000 | (value << 16 | value << 8 | value);
    }
}

```

第三步, 根据输入参数 N , 寻找前 N 个信号最强的圆心

```

results = new int[accSize * 3];
int[] output = new int[width * height];

// 获取最大的前accSize个值
for (int x = 0; x < width; x++) {
    for (int y = 0; y < height; y++) {
        int value = (acc[x + (y * width)] & 0xff);

        // if its higher than lowest value add it and then sort
        if (value > results[(accSize - 1) * 3]) {
            // add to bottom of array
            results[(accSize - 1) * 3] = value; // 像素值
            results[(accSize - 1) * 3 + 1] = x; // 坐标x
            results[(accSize - 1) * 3 + 2] = y; // 坐标y

            // shift up until its in right place
            int i = (accSize - 2) * 3;
            while ((i >= 0) && (results[i + 3] > results[i])) {
                for (int j = 0; j < 3; j++) {
                    int temp = results[i + j];
                    results[i + j] = results[i + 3 + j];
                    results[i + 3 + j] = temp;
                }
                i = i - 3;
            }
            if (i < 0)
                break;
        }
    }
}

```

13.5 图像金字塔

第四步, 根据圆心与半径, 转换到平面坐标画出圆, 代码如下:

```

// 根据找到的半径R, 中心点像素坐标p(x, y), 在原图像上绘制圆
System.out.println("top " + accSize + " matches:");
for (int i = accSize - 1; i >= 0; i--) {

```

```
drawCircle(results[i * 3], results[i * 3 + 1], results[i * 3 + 2]);
}
```

其中方法 `drawCircle()` 的代码如下：

```
pix = 250; // 颜色值，默认为白色
```

```
int x, y, r2;
int radius = r;
r2 = r * r;
```

```
// 绘制圆的上下左右四个点
```

```
setPixel(pix, xCenter, yCenter + radius);
setPixel(pix, xCenter, yCenter - radius);
setPixel(pix, xCenter + radius, yCenter);
setPixel(pix, xCenter - radius, yCenter);
```

```
y = radius;
```

```
x = 1;
```

```
y = (int) (Math.sqrt(r2 - 1) + 0.5);
```

```
// 边缘填充算法，其实可以直接通过循环所有像素、计算到中心点的距离来做
```

```
// 这个方法是别人写的，超赞！
```

```
while (x < y) {
    setPixel(pix, xCenter + x, yCenter + y);
    setPixel(pix, xCenter + x, yCenter - y);
    setPixel(pix, xCenter - x, yCenter + y);
    setPixel(pix, xCenter - x, yCenter - y);
    setPixel(pix, xCenter + y, yCenter + x);
    setPixel(pix, xCenter + y, yCenter - x);
    setPixel(pix, xCenter - y, yCenter + x);
    setPixel(pix, xCenter - y, yCenter - x);
    x += 1;
    y = (int) (Math.sqrt(r2 - x * x) + 0.5);
}
```

```
if (x == y) {
    setPixel(pix, xCenter + x, yCenter + y);
    setPixel(pix, xCenter + x, yCenter - y);
    setPixel(pix, xCenter - x, yCenter + y);
    setPixel(pix, xCenter - x, yCenter - y);
}
```

第五步，输出处理后的像素数组作为图片显示，代码如下：

```
CircleHough ch = new CircleHough();
ch.init(inPixels, width, height, 40);
outPixels = ch.process();
setRGB(dest, 0, 0, width, height, outPixels);
```

3. 霍夫空间显示

在图像数组由平面坐标空间变换到霍夫空间以后，处理得到圆与直线在霍夫空间的特征

显示, 转换为图像显示, 实现的代码如下:

```
int width = src.getWidth();
int height = src.getHeight();

if ( dest == null )
    dest = createCompatibleDestImage( src, null );

int[] inPixels = new int[width*height];
getRGB( src, 0, 0, width, height, inPixels );

if(type == LINE_TYPE)
{
    LineHough lh = new LineHough();
    lh.init(inPixels, width, height);
    lh.process();
    int rmax = (int)Math.sqrt(width*width + height*height);
    BufferedImage houghImage = new BufferedImage(180, rmax, BufferedImage.TYPE_INT_ARGB);
    setRGB( houghImage, 0, 0, 180, rmax, lh.getAcc() );
    return dest;
}
else if(type == CIRCLE_TYPE)
{
    CircleHough ch = new CircleHough();
    ch.init(inPixels, width, height, 40);
    ch.process();
    setRGB( dest, 0, 0, width, height, ch.getAcc() );
    return dest;
}
else
{
    throw new IllegalArgumentException("Warning: not supported type...");
}
```

上面代码中方法 `lh.getAcc()` 即返回霍夫空间特征数组, 完整的调用霍夫直线与圆检测的代码实现参见源文件中 13-03 的 `HoughFilter.java`, 输入参数 `type` 表示圆检测或直线检测。基于霍夫变换实现圆检测原理方法, 本节就介绍到这里, 感兴趣的读者可以在此基础上将输入圆半径从指定值变为最大最小值范围, 继续修改该程序, 运行实现自己的版本。

13.5 图像金字塔

图像金字塔可以看成一系列分辨率从高到低的图像集合组成, 最下层的图像分辨率最高, 最上层的图像分辨最低, 但是图像内容相同。图像金字塔在图像压缩、视觉检测、图像融合等方面都有实际应用价值与意义。

1. 金字塔概述

对一张图像不断地模糊之后向下采样, 得到不同分辨率的图像, 同时每次得到的新的

图像宽与高是原来图像的 1/2，最常见就是基于高斯的模糊之后采样，得到的一系列图像称为高斯金字塔。上述这个过程称为金字塔的 REDUCE 过程，直到得到你想要的金字塔级数 ($G_0, G_1, G_2 \cdots G_n$)。REDUCE 过程是由两步组成的，分别是高斯模糊与偶数行采样，公式表示如下：

$$G_{n+1}(i, j) = \sum_{m=-2}^2 \sum_{n=-2}^2 W(m, n) G_n(2i - m, 2j - n)$$

其中 $W(m, n) = W(m) \times W(n)$ 表示 5×5 的高斯卷积核 $W(r) = \left\{ \frac{1}{4} - \frac{a}{2}, \frac{1}{4}, a, \frac{1}{4}, \frac{1}{4} - \frac{a}{2} \right\}$ ，常见的 a 的取值范围为 $[0.3, 0.6]$ 。高斯金字塔还可以通过第 G_n 层得到第 G_{n-1} 层、由第 G_{n-1} 得到 G_{n-2} 层……直到 G_0 ，该过程称为高斯金字塔的 EXPAND 操作，同样公式表示如下：

$$\text{EXPAND}(G_n) = 4 \sum_{m=-2n}^2 \sum_{n=-2}^2 W(m, n) G_n\left(\frac{(i-m)}{2}, \frac{(j-m)}{2}\right)$$

在 expand 操作的上述定义中必须保证 $\frac{(i-m)}{2}$ 与 $\frac{(j-m)}{2}$ 为整除，这在后续的编程实现中将会做更加详细的描述。假设有一幅数字图像 T ，其最初的第 0 层高斯金字塔 G_0 就是它本身，根据 G_0 可以获得其下一级图像金字塔 G_1 ，则可以得到第 0 层的拉普拉斯金字塔为 $L_0 = G_0 - \text{EXPAND}(G_1)$ ，以此类推，可以获得图像完整的拉普拉斯金字塔。完整的关系如图 13-7 所示。

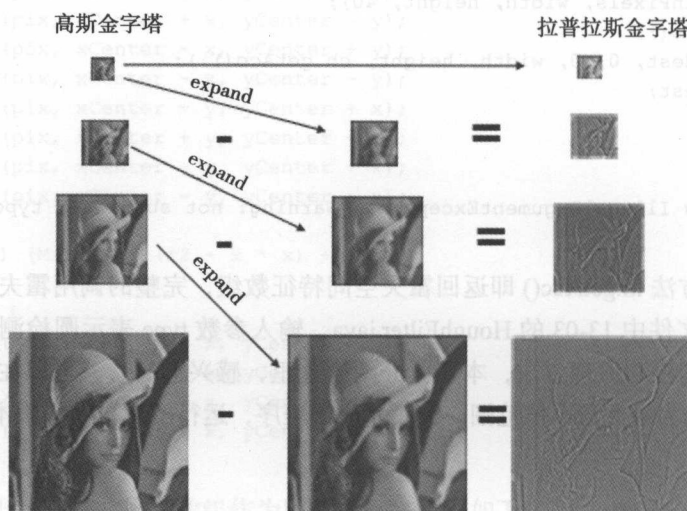


图 13-7 高斯金字塔演示

2. 高斯不同 (DOG)

高斯不同 (Difference Of Gaussian) 在模糊图像重建、图像锐化、角点检测中都有应用，高斯不同又称为高斯函数差分，是指同一幅图像用不同半径的高斯核函数模糊之后求取差值得到，即对图像用不同的高斯窗口进行低通滤波，然后相减得到的差值。当两

个高斯窗口取值比率(1.6)适当时可以看成LOG算子的近似值,而使用上面提到的公式 $L_0 = G_0 - EXPAND(G_1)$ 进行推导从其结果就可以近似看出是高斯不同(DOG)。关于高斯不同与拉普拉斯高斯之间的关系与联系,其数学推导与证明是一个很长的话题,感兴趣的读者可以自己阅读相关资料,这里就不再赘述。

3. 实现

编程实现高斯金字塔的REDUCE过程与EXPAND过程,以及高斯分差(DOG)的思路大致如下:

- 1) 首先实现一个 5×5 窗口的高斯模糊类。
- 2) 根据输入参数确定金字塔级数(高度),进行REDUCE操作建立高斯金字塔。
- 3) 对高斯金字塔的每一层完成EXPAND操作。
- 4) 根据第2、3步的结果,得到拉普拉斯高斯金字塔。

根据上述思路,首先第一步需要编程实现对图像的高斯模糊,采用的高斯模糊核算子为

$W(r) = \left\{ \frac{1}{4} - \frac{a}{2}, \frac{1}{4}, a, \frac{1}{4}, \frac{1}{4} - \frac{a}{2} \right\}$, 基于该一维高斯模糊算子,实现一维高斯模糊方法代码如下:

```
private void blur(int[] inPixels, int[] outPixels, int width, int height)
{
    int subCol = 0;
    int index = 0, index2 = 0;
    float redSum=0, greenSum=0, blueSum=0;
    for(int row=0; row<height; row++) {
        int ta = 0, tr = 0, tg = 0, tb = 0;
        for(int col=0; col<width; col++) {
            // index = row * width + col;
            redSum=0;
            greenSum=0;
            blueSum=0;
            for(int m=-2; m<=2; m++) {
                subCol = col + m;
                if(subCol < 0 || subCol >= width) {
                    subCol = 0;
                }
                index2 = row * width + subCol;
                ta = (inPixels[index2] >> 24) & 0xff;
                tr = (inPixels[index2] >> 16) & 0xff;
                tg = (inPixels[index2] >> 8) & 0xff;
                tb = inPixels[index2] & 0xff;
                redSum += (tr * gaussianKeneral[m + 2]);
                greenSum += (tg * gaussianKeneral[m + 2]);
                blueSum += (tb * gaussianKeneral[m + 2]);
            }
            outPixels[index] = (ta << 24) | (clamp(redSum) << 16) |
                (clamp(greenSum) << 8) | clamp(blueSum);
            index += height;
        }
    }
}
```

调用该方法，实现图像水平与竖直方向高斯模糊的代码如下：

```
int width = src.getWidth();
int height = src.getHeight();

if ( dest == null )
    dest = createCompatibleDestImage( src, null );

int[] inPixels = new int[width*height];
int[] outPixels = new int[width*height];
getRGB( src, 0, 0, width, height, inPixels);
blur( inPixels, outPixels, width, height); // H Gaussian
blur( outPixels, inPixels, height, width); // V Gaussain
setRGB(dest, 0, 0, width, height, inPixels );
return dest;
```

第二步，基于第 n 层图像高斯模糊结果完成图像下采样，得到高斯金字塔的第 $n+1$ 层的代码实现如下：

```
public BufferedImage[] pyramidDown(BufferedImage src) {
    BufferedImage[] imagePyramids = new BufferedImage[level + 1];
    imagePyramids[0] = src;
    whData = new int[level][2];
    whData[0][0] = src.getWidth();
    whData[0][1] = src.getHeight();
    for(int i=1; i<imagePyramids.length; i++) {
        imagePyramids[i] = pyramidReduce(imagePyramids[i-1]);
        if(i < level) {
            whData[i][0] = imagePyramids[i].getWidth();
            whData[i][1] = imagePyramids[i].getHeight();
        }
    }
    return imagePyramids;
}
```

其中关键操作 REDUCE 方法实现代码如下：

```
int width = src.getWidth();
int height = src.getHeight();
BufferedImage dest = createSubCompatibleDestImage(src, null);
int[] inPixels = new int[width*height];
int ow = width/2;
int oh = height/2;
int[] outPixels = new int[ow*oh];
getRGB(src, 0, 0, width, height, inPixels );
int inRow=0, inCol = 0, index = 0, oudex =0, ta = 0;
float[][] keneralData = this.getHVGaussianKeneral();
for(int row=0; row<oh; row++) {
```

```

for(int col=0; col<ow; col++){
    inRow = 2* row;
    inCol = 2* col;
    if(inRow >= height) {
        inRow = 0;
    }
    if(inCol >= width) {
        inCol = 0;
    }
    float sumRed = 0, sumGreen = 0, sumBlue = 0;
    for(int subRow = -2; subRow <= 2; subRow++){
        int inRowOff = inRow + subRow;
        if(inRowOff >= height || inRowOff < 0) {
            inRowOff = 0;
        }
        for(int subCol = -2; subCol <= 2; subCol++){
            int inColOff = inCol + subCol;
            if(inColOff >= width || inColOff < 0) {
                inColOff = 0;
            }
            index = inRowOff * width + inColOff;
            ta = (inPixels[index] >> 24) & 0xff;
            int red = (inPixels[index] >> 16) & 0xff;
            int green = (inPixels[index] >> 8) & 0xff;
            int blue = inPixels[index] & 0xff;
            sumRed += kernalData[subRow + 2][subCol + 2] * red;
            sumGreen += kernalData[subRow + 2][subCol + 2] * green;
            sumBlue += kernalData[subRow + 2][subCol + 2] * blue;
        }
    }
    outhex = row * ow + col;
    outPixels[outhex] = (ta << 24) | (clamp(sumRed) << 16) | (clamp(sumGreen) << 8) | clamp(sumBlue);
}
setRGB( dest, 0, 0, ow, oh, outPixels );
return dest;

```

第三步，基于建立的高斯金字塔实现图像金字塔的 EXPAND 操作，代码如下：

```

public BufferedImage[] pyramidUp(BufferedImage[] srcImage) {
    BufferedImage[] imagePyramids = new BufferedImage[srcImage.length-1];
    for(int i=1; i<srcImage.length; i++) {
        imagePyramids[i-1] = pyramidExpand(srcImage[i], i-1);
    }
    return imagePyramids;
}

```

该方法要求输入参数为高斯金字塔的每一层图像，其中实现金字塔 EXPAND 操作的代码如下：

```

public BufferedImage pyramidExpand(BufferedImage src, int levelIndex) {
    int width = src.getWidth();
    int height = src.getHeight();
    System.out.println("expand src.width = " + width + " , src.height = " + height);
    int[] inPixels = new int[width*height];
    getRGB(src, 0, 0, width, height, inPixels );
    int ow = whData[levelIndex][0];
    int oh = whData[levelIndex][1];
    System.out.println("expand exp.width = " + ow + " , exp.height = " + oh);
    int[] outPixels = new int[ow * oh];
    int index = 0, outdex = 0, ta = 0;
    float[][] kernalData = this.getHVGaussianKeneral();
    BufferedImage dest = createTwiceCompatibleDestImage(src, null, levelIndex);
    for(int row=0; row<oh; row++) {
        for(int col=0; col<ow; col++) {
            float sumRed = 0, sumGreen = 0, sumBlue = 0;
            for(int subRow = -2; subRow <= 2; subRow++) {
                double srcRow = (row + subRow)/2.0;
                double j = Math.floor(srcRow);
                double t = srcRow - j;
                if(t > 0) {
                    continue;
                }
                if(srcRow >= height || srcRow < 0) {
                    srcRow = 0;
                }
                for(int subCol = -2; subCol <= 2; subCol++) {
                    double srcColOff = (col + subCol)/2.0;
                    j = Math.floor(srcColOff);
                    t = srcColOff - j;
                    if(t > 0) {
                        continue;
                    }
                    if(srcColOff >= width || srcColOff < 0) {
                        srcColOff = 0;
                    }
                    index = (int)(srcRow * width + srcColOff);
                    ta = (inPixels[index] >> 24) & 0xff;
                    int red = (inPixels[index] >> 16) & 0xff;
                    int green = (inPixels[index] >> 8) & 0xff;
                    int blue = inPixels[index] & 0xff;
                    sumRed += kernalData[subRow + 2][subCol + 2] * red;
                    sumGreen += kernalData[subRow + 2][subCol + 2] * green;
                    sumBlue += kernalData[subRow + 2][subCol + 2] * blue;
                }
            }
            outdex = row * ow + col;
            outPixels[outdex] = (ta << 24) | (clamp(4.0f * sumRed) << 16) |
                (clamp(4.0f * sumGreen) << 8) | clamp(4.0f * sumBlue);
            // outPixels[outdex] = (ta << 24) | (clamp(sumRed) << 16) |
                (clamp(sumGreen) << 8) | clamp(sumBlue);
        }
    }
}

```

```

    }
    setRGB( dest, 0, 0, ow, oh, outPixels );
    return dest;
}

```

第四步，计算图像的拉普拉斯金字塔，实现代码如下：

```

public BufferedImage[] getLaplacianPyramid(BufferedImage[] reduceImages) {
    BufferedImage[] laplaciImages = new BufferedImage[reduceImages.length - 1];
    for(int i=1; i<reduceImages.length; i++) {
        BufferedImage expandImage = pyramidExpand(reduceImages[i], i-1);
        laplaciImages[i-1] = createCompatibleDestImage(expandImage, null);
        int width = reduceImages[i-1].getWidth();
        int height = reduceImages[i-1].getHeight();

        int ewidth = expandImage.getWidth();
        width = (width > ewidth) ? ewidth : width;
        height = (height > expandImage.getHeight()) ? expandImage.getHeight() : height;
        System.out.println(" width = " + width + " expand width = " + ewidth);

        int[] reducePixels = new int[width*height];
        int[] expandPixels = new int[width*height];
        int[] laPixels = new int[width*height];
        getRGB( reduceImages[i-1], 0, 0, width, height, reducePixels);
        getRGB( expandImage, 0, 0, width, height, expandPixels );
        int index = 0;
        int er = 0, eg = 0, eb = 0;
        for(int row=0; row<height; row++) {
            int ta = 0, tr = 0, tg = 0, tb = 0;
            for(int col=0; col<width; col++) {
                index = row * width + col;
                ta = (reducePixels[index] >> 24) & 0xff;
                tr = (reducePixels[index] >> 16) & 0xff;
                tg = (reducePixels[index] >> 8) & 0xff;
                tb = reducePixels[index] & 0xff;

                ta = (expandPixels[index] >> 24) & 0xff;
                er = (expandPixels[index] >> 16) & 0xff;
                eg = (expandPixels[index] >> 8) & 0xff;
                eb = expandPixels[index] & 0xff;

                tr = (tr - er);
                tg = (tg - eg);
                tb = (tb - eb);

                laPixels[index] = (ta << 24) | (clamp(tr) << 16) |
                    (clamp(tg) << 8) | clamp(tb);
            }
        }
        setRGB( laplaciImages[i-1], 0, 0, width, height, laPixels );
    }
}

```



```

    }

    return laplacilImages;
}

```

完整的图像金字塔实现源代码参见源文件的 13-05 中源代码文件 PyramidAlgorithm.java 与 GaussianFilter.java，测试类为 PyramidDemoUI.java 源代码文件，关于图像金字塔与代码实现，本节就介绍到这里，源代码也是本书内容的一部分，阅读与运行本节代码有助于读者更好地理解本节内容。

13.6 Harris 角度检测

Harris 角度检测是通过数学计算在图像上发现角度特征的一种算法，而且其具有旋转不变性的特质。基于 Harris 角度检测改进的算法——Shi-Tomasi 角度检测，在很多领域都有应用。

1. 基本原理

Harris 角度检测通过寻找本地响应最大化关键点来实现角点检测，而对一幅图像来说，角度是它最明显与重要的特征之一，基于图像一阶导数处理，角度在各个方向的变化是最大的，而边缘区域只在某一方向有明显变化，一个直观的图示见图 13-8。

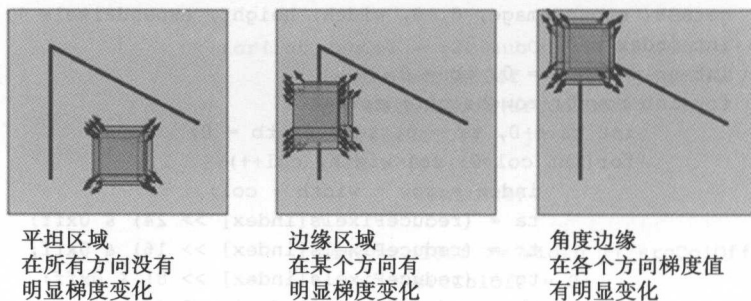


图 13-8 边缘效果

根据上述原理，Harris 角度检测就变成寻找本地像素点在 X 与 Y 方向的最大响应，公式表示如下：

$$E(u, v) = \sum_{x, y} W(x, y) [I(x + u, y + v) - I(x, y)]^2$$

其中 $W(x, y)$ 是窗口函数， $I(x, y)$ 表示像素点强度值，常见为图像灰度值。上述公式经过一系列的数学推导，最终可以得到如下公式：

$$E(u, v) = [u, v] \left(\sum \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} \right) \begin{bmatrix} u \\ v \end{bmatrix}, \text{ 当值 } u, v \text{ 较小时近似为 } E(u, v) \cong [u, v] M \begin{bmatrix} u \\ v \end{bmatrix}$$

其中 M 是根据图像 X 方向、Y 方向、XY 方向计算导数得到的 2×2 的矩阵，表示为：

$$M = \sum_{x,y} W(x,y) \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}$$

其中 $W(x, y)$ 为权重函数, 常采用高斯权重, 而后面的矩阵称为图像像素的 Harris 矩阵。最终根据 Harris 矩阵计算出矩阵特征值 λ_1 、 λ_2 , 然后计算出 Harris 角度响应值 R , 计算公式如下:

$$R = \det M - k(\text{trace } M)^2$$

其中 $\det M = \lambda_1 \lambda_2$, $\text{trace } M = \lambda_1 + \lambda_2$, 系数 k 常见的取值范围为 $[0.04, 0.06]$ 。关于 Harris 角度检测数学知识的简单介绍就到这里为止, 一切并不复杂。如果读者想了解更多关于 Harris 角度检测的数学推导与原理, 可以阅读相关文献与资料。从上面的介绍读者不难看出, Harris 角度检测的关键在于计算角度响应值 R 。

2. 实现

基于 Harris 角度检测的原理, 编码实现 Harris 角度检测的大致步骤如下:

- 1) 计算图像 X 方向与 Y 方向的一阶高斯偏导数 I_x 与 I_y 。
- 2) 根据第一步结果得到 I_x^2 、 I_y^2 与 $I_x I_y$ 的值。
- 3) 高斯模糊第二步三个值得到 S_{xx} 、 S_{yy} 与 S_{xy} 。
- 4) 定义每个像素的 Harris 矩阵, 计算出矩阵的两个特质值。
- 5) 计算出每个像素的 R 值。
- 6) 使用 3×3 或 5×5 的窗口, 实现简单的非最大值压制。
- 7) 根据角度检测结果计算, 以绿色标记提取到的关键点, 显示在原图上。

在上述实现中, 权重采用了高斯函数, 第三步是可选步骤, 此外还可以采用 Sobel 算子计算 I_x^2 、 I_y^2 与 $I_x I_y$, 这样可以简化第一步与第二步的计算。根据上面的步骤, 实现代码如下:

```
int width = src.getWidth();
int height = src.getHeight();
initSettings(height, width);
if (dest == null)
    dest = createCompatibleDestImage( src, null );

BufferedImage grayImage = super.filter(src, null);
int[] inPixels = new int[width*height];

// first step - Gaussian first-order Derivatives (3 × 3) - X - gradient, (3 × 3)
// - Y - gradient
filter.setDirectionType(GaussianDerivativeFilter.X_DIRECTION);
BufferedImage xImage = filter.filter(grayImage, null);
getRGB( xImage, 0, 0, width, height, inPixels );
extractPixelData(inPixels, GaussianDerivativeFilter.X_DIRECTION, height, width);

filter.setDirectionType(GaussianDerivativeFilter.Y_DIRECTION);
BufferedImage yImage = filter.filter(grayImage, null);
getRGB( yImage, 0, 0, width, height, inPixels );
extractPixelData(inPixels, GaussianDerivativeFilter.Y_DIRECTION, height, width);
```

```
// second step - calculate the Ix^2, Iy^2 and Ix*Iy
for(HarrisMatrix hm : harrisMatrixList)
{
    double Ix = hm.getXGradient();
    double Iy = hm.getYGradient();
    hm.setIxIy(Ix * Iy);
    hm.setXGradient(Ix*Ix);
    hm.setYGradient(Iy*Iy);
}
```

// 基于高斯方法，中心点化窗口计算一阶导数和，关键一步 SumIx2, SumIy2 and SumIxIy，高斯模糊
calculateGaussianBlur(width, height);

// 求取Harris Matrix 特征值

// 计算角度相应值 $R = \text{Det}(H) - \lambda \cdot (\text{Trace}(H))^2$
harrisResponse(width, height);

// based on R, compute non-max suppression
nonMaxValueSuppression(width, height);

// match result to original image and highlight the key points
int[] outPixels = matchToImage(width, height, src);

// return result image
setRGB(dest, 0, 0, width, height, outPixels);
return dest;

上述代码中，GaussianDerivativeFilter 类实现了对图像 X 方向和 Y 方向的一阶导数与二阶导数计算和结果输出，默认的高斯窗口函数为 3×3 大小、 $\sigma = 10$ 。选择不同的参数，可实现对 X 方向、Y 方向与 XY 方向的导数计算。这样就完成了第一步的功能。第三步的代码实现对应方法为 calculateGaussianBlur()，其目的是进一步降低噪声影响，实现代码如下：

```
int index = 0;
int radius = (int>window_radius;
double[][] gw = get2DKernalData(radius, sigma);
double sumxx = 0, sumyy = 0, sumxy = 0;
for(int row=0; row<height; row++) {
    for(int col=0; col<width; col++) {
        for(int subrow =-radius; subrow<=radius; subrow++)
        {
            for(int subcol=-radius; subcol<=radius; subcol++)
            {
                int nrow = row + subrow;
                int ncol = col + subcol;
                if(nrow >= height || nrow < 0)
                {
                    nrow = 0;
                }
                if(ncol >= width || ncol < 0)
                {

```

```

        ncol = 0;
    }
    int index2 = nrow * width + ncol;
    HarrisMatrix whm = harrisMatrixList.get(index2);
    sumxx += (gw[subrow + radius][subcol + radius] * whm.
        getXGradient());
    sumyy += (gw[subrow + radius][subcol + radius] * whm.
        getYGradient());
    sumxy += (gw[subrow + radius][subcol + radius] * whm.
        getIxIy());
    }
    index = row * width + col;
    HarrisMatrix hm = harrisMatrixList.get(index);
    hm.setXGradient(sumxx);
    hm.setYGradient(sumyy);
    hm.setIxIy(sumxy);

    // clean up for next loop
    sumxx = 0;
    sumyy = 0;
    sumxy = 0;
}
}

```

第四步与第五步，实现矩阵特征值计算与 R 值求取的代码如下：

```

private void harrisResponse(int width, int height) {
    int index = 0;
    for(int row=0; row<height; row++) {
        for(int col=0; col<width; col++) {
            index = row * width + col;
            HarrisMatrix hm = harrisMatrixList.get(index);
            double c = hm.getIxIy() * hm.getIxIy();
            double ab = hm.getXGradient() * hm.getYGradient();
            double aplusb = hm.getXGradient() + hm.getYGradient();
            double response = (ab - c) - lambda * Math.pow(aplusb, 2);
            hm.setR(response);
        }
    }
}

```

第六步，使用 3×3 窗口，实现简单地非最大信号压制的代码如下：

```

int index = 0;
int radius = (int>window_radius;
for(int row=0; row<height; row++) {
    for(int col=0; col<width; col++) {
        index = row * width + col;
        HarrisMatrix hm = harrisMatrixList.get(index);
        double maxR = hm.getR();
        boolean isMaxR = true;
    }
}

```

```

        for(int subrow ==-radius; subrow<=radius; subrow++)
        {
            for(int subcol==radius; subcol<=radius; subcol++)
            {
                int nrow = row + subrow;
                int ncol = col + subcol;
                if(nrow >= height || nrow < 0)
                {
                    nrow = 0;
                }
                if(ncol >= width || ncol < 0)
                {
                    ncol = 0;
                }
                int index2 = nrow * width + ncol;
                HarrisMatrix hmr = harrisMatrixList.get(index2);
                if(hmr.getR() > maxR)
                {
                    isMaxR = false;
                }
            }
        }
        if(isMaxR)
        {
            hm.setMax(maxR);
        }
    }
}

```

第七步，最终在图像上显示结果数据的代码如下：

```

int[] inPixels = new int[width*height];
int[] outPixels = new int[width*height];
getRGB( src, 0, 0, width, height, inPixels );
int index = 0;
for(int row=0; row<height; row++) {
    int ta = 0, tr = 0, tg = 0, tb = 0;
    for(int col=0; col<width; col++) {
        index = row * width + col;
        ta = (inPixels[index] >> 24) & 0xff;
        tr = (inPixels[index] >> 16) & 0xff;
        tg = (inPixels[index] >> 8) & 0xff;
        tb = inPixels[index] & 0xff;
        HarrisMatrix hm = harrisMatrixList.get(index);
        if(hm.getMax() > 0)
        {
            tr = 0;
            tg = 255; // make it as green for corner key pointers
            tb = 0;
            outPixels[index] = (ta << 24) | (tr << 16) | (tg << 8) | tb;
        }
    }
}

```

```

else
{
    outPixels[index] = (ta << 24) | (tr << 16) | (tg << 8) | tb;
}

}

return outPixels;

```

Harris 角度检测完整源代码实现对应类 `HarrisCornerDetector`，其继承了 `GrayFilter` 类，目的是实现图像灰度化，类 `HarrisMatrix` 是用来存储计算 Harris 矩阵的数据结构的。类 `GaussianDerivativeFilter` 则是关于图像的二维高斯一阶、二阶导数的代码实现，高斯公式相关数学知识前面章节已经介绍过，二维高斯的偏导数可以参见本书相应章节的源代码学习。如何计算 2×2 矩阵特征值同样可以在本书附录 A 中找到。

上述提到的 Harris 角度检测相关的类实现代码均可以在本书源文件中的 13-06 找到，阅读与运行测试 Harris 角度检测的代码实现有助于读者进一步理解该算法，感兴趣的读者可以进一步改写与优化代码实现。此外，很多图像处理软件都会用标记来显示关键点像素，这没有在代码里实现，只是将关键点像素改为了绿色，所以读者还可以完成自己的关键点像素标记显示。

13.7 SIFT 特征提取

SIFT 算法的历史可追溯到 1999 年，当年，Lower 为解决 Harris 角度检测对图像尺度变化敏感的问题，提出了局部特征描述子算法，2004 年在他人的基础上，Lower 又进一步完善了该算法。SIFT (Scale Invariant Feature Transform) 算法主要是解决图像特征在尺度空间不变性的问题，通过寻找图像上特征关键点建立最终图像的本地特征描述子。SIFT 算法在图像特征匹配、搜索、视频跟踪等领域都有很重要的应用。SIFT 特征算子是图像的局部特征，其对旋转、放缩、亮度变化保持不变，对视角变化、仿射变换、噪声等也不是十分敏感。完整的 SIFT 算法是由一系列不同的图像操作按照一定顺序组成的，最终得到特征描述子。这其中，首先要建立尺度空间计算高斯差分 (DOG)，然后在此基础上实现极值查找与非关键点过滤，并根据尺度参数对得到的关键点，实现子区域梯度与角度计算，建立方向直方图，最后才能根据关键点建立关键点描述子。

1. 尺度空间建立

尺度空间建立，基于高斯核函数 σ 取值的不同来实现，其中最重要的是建立高斯金字塔，对每一层基于不同的 σ 值构建尺度图像集合，一般来说 σ 值越大，图像越模糊。然后基于金字塔每一层图像集合，构建得到高斯差分图像 (DOG)，高斯差分图像具有放缩不变性特征，这样就消除了图像放缩对特征提取的影响。关于建立尺度空间建立的图示见图 13-9。

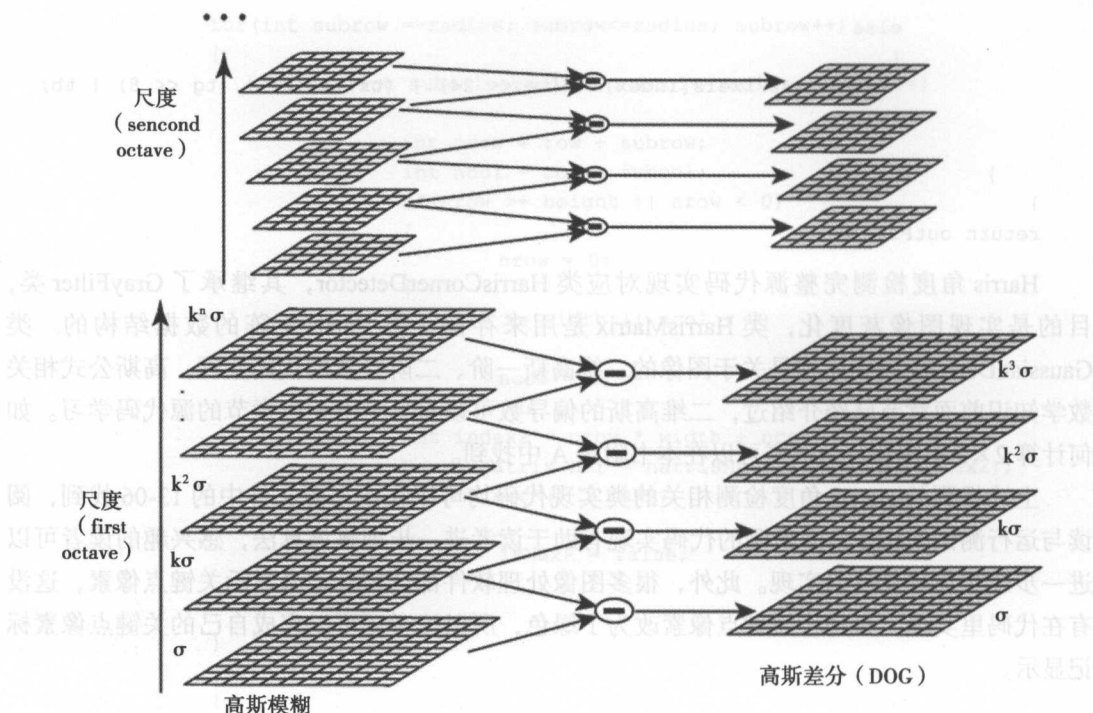


图 13-9 尺度空间

计算高斯差分的公式为：

$$D(x, y, \sigma) = (G(x, y, k\sigma) - G(x, y, \sigma)) * I(x, y) \\ = L(x, y, k\sigma) - L(x, y, \sigma)$$

其中 L 表示高斯模糊图像。上述尺度的每一层英文有个很专业的名称 Octave，中文的翻译简直是五花八门，所以在这里还是决定保留其英文原称。根据研究表明，当 $\sigma = 1.6$ 、 $k = 2^{\frac{1}{4}}$ 时，效果近似等于拉普拉斯 (LOG) 的结果。根据上述输入参数，建立尺度空间还必须解决的另外一个问题是尺度层数问题，SIFT 算法论文中给出的是 $s+3$ 层，其中 $K^s = 2$ ，这样下采样得到下一个 Octave 是取当前 Octave 尺度参数等于 2σ 对应的第 s 层。上述这些参数及采样层数是决定尺度空间建立的必要前提，根据这些参数完整实现尺度空间建立与高斯差分 (DOG) 的代码类为 ScaleOctave.java，其中关键的方法 buildStub() 是实现建立下采样对应的层图像数据，build() 方法实现图像每个 Octave 尺度空间的建立。同时为了方便操作像素数组，定义了一个数据结构类 Float2DArray.java，用来存储图像像素数据及其操作结果。基于 ScaleOctave 类实现 SIFT 算法尺度空间建立的算法代码如下：

```
// build scale space of octave
Float2DArray next;
for ( int i = 0; i < octaves.length; ++i )
{
```

```

octaves[ i ] = new ScaleOctave(
    blurredData,
    sigma,
    sigma_diff,
    kernel_diff );
octaves[ i ].buildStub();
next = new Float2DArray(
    blurredData.width / 2 + blurredData.width % 2,
    blurredData.height / 2 + blurredData.height % 2 );
GaussianUtil.downsample( octaves[ i ].getLevel( 1 ), next );
if ( blurredData.width > max_size || blurredData.height > max_size )
    octaves[ i ] = null;
blurredData = next;
}

```

其中 GaussianUtil.java 是工具类, 提供了高斯核函数生成、归一化、像素数组高斯卷积、梯度计算等常见的图像处理功能, 相信这些对本书读者都不再陌生。

2. 极值寻找与过滤

对于得到 DOG 的图像, 在每个 Octave 的每个尺度 (Scale) 上, 以及它的上下两个尺度上, 使用 3×3 的窗口大小可寻找极大值或极小值, 如图 13-10 所示。

这里, X 表示中心像素, 圆形表示周围 26 个像素点, 这样就得到了极值。获取到极值之后, 因为是在离散空间采样, 并不能保证中心位置就是极值的准确位置, 所以对于任意一个上述空间像素点 $D(x, y, \sigma)$, 算法作者在 2002 年提出通过在三维空间泰勒级数展开二次方程插值实现极大值定位查找, 这样可以排除非极值关键点, 规定在任意一个方向上移动 0.5 以上就重新计算位置, 如果成功得到新位置坐标 $D(x', y', \sigma')$ 则表示成功得到极值位置。该公式表示为:

$$\hat{x} = -\frac{\partial^2 D^{-1}}{\partial X^2} \frac{\partial D}{\partial X}$$

其中 $X = (x, y, \sigma)^T$ 表示从当前采样点的偏移。

该公式的对采样点进行二阶偏导数计算, 前面章节中已经讲过如何计算图像一阶、二阶偏导数, 因为图像都是二维的, 这里唯一不同的是要在三维空间计算, 计算得到二阶偏导数本质上是一个三维空间的拉普拉斯算子, 然后使用一阶偏导数的值相乘, 就变成了两个矩阵相乘, 得到的结果就是 x, y, σ 上的偏移量, 根据偏移量带入计算得到新位置。根据如下公式:

$$D(\hat{x}) = D + \frac{1}{2} \frac{\partial D^T}{\partial X} \hat{x}$$

如果计算得到的 $|D(\hat{x})|$ 值小于 0.025, 则该极值点应该被丢弃, 反之则保留。对保留的极值点进行 Harris 矩阵计算, 如果值大于阈值则丢弃, 这样就进一步消除了边缘对关键点提

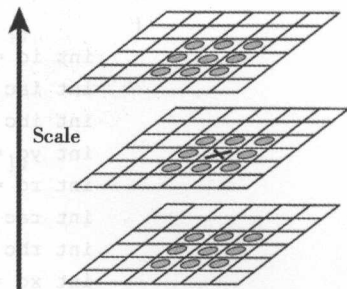


图 13-10 极大值寻找

取的影响。该步骤也是 SIFT 算法关键步骤之一，有助于提高 SIFT 算法精确度。极大值的寻找与过滤也被封装为一个单独类，即 `OctaveKeyPointersDetector.java`。其中 `detectCandidates()` 方法实现上述全部功能，代码实现如下，请读者对照原理解释阅读，这有助于读者更好地理解该步骤。

```
Float2DArray[] d = octave.getDoGs();
```

```
for ( int i = d.length - 2; i >= 1; --i )
```

```
{
```

```
    int ia = i - 1;
```

```
    int ib = i + 1;
```

```
    for ( int y = d[ i ].height - 2; y >= 1; --y )
```

```
    {
```

```
        int r = y * d[ i ].width;
```

```
        int ra = r - d[ i ].width;
```

```
        int rb = r + d[ i ].width;
```

```
        X : for ( int x = d[ i ].width - 2; x >= 1; --x )
```

```
        {
```

```
            int ic = i;
```

```
            int iac = ia;
```

```
            int ibc = ib;
```

```
            int yc = y;
```

```
            int rc = r;
```

```
            int rac = ra;
```

```
            int rbc = rb;
```

```
            int xc = x;
```

```
            int xa = xc - 1;
```

```
            int xb = xc + 1;
```

```
            float e111 = d[ ic ].data[ r + xc ];
```

```
            // check if d(x, y, i) is an extremum
```

```
            // do it pipeline-friendly ;)
```

```
            float e000 = d[ iac ].data[ rac + xa ];
```

```
            boolean isMax = e000 < e111;
```

```
            boolean isMin = e000 > e111;
```

```
            if ( !( isMax || isMin ) ) continue;
```

```
            float e100 = d[ iac ].data[ rac + xc ];
```

```
            isMax &= e100 < e111;
```

```
            isMin &= e100 > e111;
```

```
            if ( !( isMax || isMin ) ) continue;
```

```
            float e200 = d[ iac ].data[ rac + xb ];
```

```
            isMax &= e200 < e111;
```

```
            isMin &= e200 > e111;
```

```

if ( !( isMax || isMin ) ) continue;

float e010 = d[ iac ].data[ rc + xa ];
isMax &= e010 < e111;
isMin &= e010 > e111;
if ( !( isMax || isMin ) ) continue;
float e110 = d[ iac ].data[ rc + xc ];
isMax &= e110 < e111;
isMin &= e110 > e111;
if ( !( isMax || isMin ) ) continue;
float e210 = d[ iac ].data[ rc + xb ];
isMax &= e210 < e111;
isMin &= e210 > e111;
if ( !( isMax || isMin ) ) continue;

float e020 = d[ iac ].data[ rbc + xa ];
isMax &= e020 < e111;
isMin &= e020 > e111;
if ( !( isMax || isMin ) ) continue;
float e120 = d[ iac ].data[ rbc + xc ];
isMax &= e120 < e111;
isMin &= e120 > e111;
if ( !( isMax || isMin ) ) continue;
float e220 = d[ iac ].data[ rbc + xb ];
isMax &= e220 < e111;
isMin &= e220 > e111;
if ( !( isMax || isMin ) ) continue;

float e001 = d[ ic ].data[ rac + xa ];
isMax &= e001 < e111;
isMin &= e001 > e111;
if ( !( isMax || isMin ) ) continue;
float e101 = d[ ic ].data[ rac + xc ];
isMax &= e101 < e111;
isMin &= e101 > e111;
if ( !( isMax || isMin ) ) continue;
float e201 = d[ ic ].data[ rac + xb ];
isMax &= e201 < e111;
isMin &= e201 > e111;
if ( !( isMax || isMin ) ) continue;

float e011 = d[ ic ].data[ rc + xa ];
isMax &= e011 < e111;
isMin &= e011 > e111;
if ( !( isMax || isMin ) ) continue;

```

```

float e211 = d[ ic ].data[ rc + xb ];
isMax &= e211 < e111;
isMin &= e211 > e111;
if ( !( isMax || isMin ) ) continue;

float e021 = d[ ic ].data[ rbc + xa ];
isMax &= e021 < e111;
isMin &= e021 > e111;
if ( !( isMax || isMin ) ) continue;
float e121 = d[ ic ].data[ rbc + xc ];
isMax &= e121 < e111;
isMin &= e121 > e111;
if ( !( isMax || isMin ) ) continue;
float e221 = d[ ic ].data[ rbc + xb ];
isMax &= e221 < e111;
isMin &= e221 > e111;
if ( !( isMax || isMin ) ) continue;

float e002 = d[ ibc ].data[ rac + xa ];
isMax &= e002 < e111;
isMin &= e002 > e111;
if ( !( isMax || isMin ) ) continue;
float e102 = d[ ibc ].data[ rac + xc ];
isMax &= e102 < e111;
isMin &= e102 > e111;
if ( !( isMax || isMin ) ) continue;
float e202 = d[ ibc ].data[ rac + xb ];
isMax &= e202 < e111;
isMin &= e202 > e111;
if ( !( isMax || isMin ) ) continue;

float e012 = d[ ibc ].data[ rc + xa ];
isMax &= e012 < e111;
isMin &= e012 > e111;
if ( !( isMax || isMin ) ) continue;
float e112 = d[ ibc ].data[ rc + xc ];
isMax &= e112 < e111;
isMin &= e112 > e111;
if ( !( isMax || isMin ) ) continue;
float e212 = d[ ibc ].data[ rc + xb ];
isMax &= e212 < e111;
isMin &= e212 > e111;
if ( !( isMax || isMin ) ) continue;

float e022 = d[ ibc ].data[ rbc + xa ];

```

```

isMax &= e022 < e111;
isMin &= e022 > e111;
if ( !( isMax || isMin ) ) continue;
float e122 = d[ ibc ].data[ rbc + xc ];
isMax &= e122 < e111;
isMin &= e122 > e111;
if ( !( isMax || isMin ) ) continue;
float e222 = d[ ibc ].data[ rbc + xb ];
isMax &= e222 < e111;
isMin &= e222 > e111;
if ( !( isMax || isMin ) ) continue;

// so it is an extremum, try to localize it with subpixel
// accuracy, if it has to be moved for more than 0.5 in at
// least one direction, try it again there but maximally 5
// times

boolean isLocalized = false;
boolean isLocalizable = true;

float dx;
float dy;
float di;

float dxx;
float dyy;
float dii;

float dxy;
float dxi;
float dyi;

float ox;
float oy;
float oi;

float od = Float.MAX_VALUE; // offset square distance

float fx = 0;
float fy = 0;
float fi = 0;

int t = 5; // maximal number of re-localizations
do
{
    --t;

```



```

// derive at (x, y, i) by center of difference
dx = ( e211 - e011 ) / 2.0f;
dy = ( e121 - e101 ) / 2.0f;
di = ( e112 - e110 ) / 2.0f;

// create hessian at (x, y, i) by laplace
float e111_2 = 2.0f * e111;
dxx = e011 - e111_2 + e211;
dyy = e101 - e111_2 + e121;
dii = e110 - e111_2 + e112;

dxy = ( e221 - e021 - e201 + e001 ) / 4.0f;
dxi = ( e212 - e012 - e210 + e010 ) / 4.0f;
dyi = ( e122 - e102 - e120 + e100 ) / 4.0f;

// invert hessian
Matrix H = new Matrix( new double[][]{
    { ( double)dxx, ( double )dxy, ( double )dxi },
    { ( double)dxy, ( double )dyy, ( double )dyi },
    { ( double)dxi, ( double )dyi, ( double )dii } }, 3, 3 );
Matrix H_inv;
try
{
    H_inv = H.inverse();
}
catch ( RuntimeException e )
{
    continue X;
}
double[][] h_inv = H_inv.getArray();

// estimate the location of zero crossing being the offset of
the extremum
ox = -( float )h_inv[ 0 ][ 0 ] * dx - ( float )h_inv[ 0 ][ 1 ] * dy
    - ( float )h_inv[ 0 ][ 2 ] * di;
oy = -( float )h_inv[ 1 ][ 0 ] * dx - ( float )h_inv[ 1 ][ 1 ] * dy
    - ( float )h_inv[ 1 ][ 2 ] * di;
oi = -( float )h_inv[ 2 ][ 0 ] * dx - ( float )h_inv[ 2 ][ 1 ] * dy
    - ( float )h_inv[ 2 ][ 2 ] * di;

float odc = ox * ox + oy * oy + oi * oi;

if ( odc < 2.0f )
{

```

```

        if ( ( Math.abs( ox ) > 0.5 || Math.abs( oy ) > 0.5 || Math.
            abs( oi ) > 0.5 ) && odc < od )
        {
            od = odc;
            xc = ( int )Math.round( ( float )xc + ox );
            yc = ( int )Math.round( ( float )yc + oy );
            ic = ( int )Math.round( ( float )ic + oi );
        }
    }
}

```

上述代码可以分为三个部分。第一部分实现的是在三维空间的偏移计算与验证。如果任意一个元素的位置都是合法的，那么该元素就是局部极值点。第二部分实现的是在三维空间的偏移计算与验证。如果任意一个元素的位置都是合法的，那么该元素就是局部极值点。第三部分实现的是在三维空间的偏移计算与验证。如果任意一个元素的位置都是合法的，那么该元素就是局部极值点。

3. 关键点方向检测

经过上面两步以后得到的关键点具有旋转不变性的特征。因此，关键点方向检测主要实现关键点的旋转不变性。实现方向检测的关键点是指圆形像素子区域内的像素计算梯度与角度，并通过角度直方图来寻找关键点方向。在计算角度直方图时，以每 10 度为一个 BIN 建立直方图，这样即使得到不同的角度值，也可以归并到同一个 BIN 中。根据抛物线原理，如果知道它左右值，就可以找到它的峰值。作为该关键点的角度值，这样就完成了关键点方向检测。

同时根据 SIFT 算法计算梯度的公式，对每个关键点周围的 8 个像素点计算梯度。然后根据 SIFT 算法计算梯度的公式，对每个关键点周围的 8 个像素点计算梯度。然后根据 SIFT 算法计算梯度的公式，对每个关键点周围的 8 个像素点计算梯度。然后根据 SIFT 算法计算梯度的公式，对每个关键点周围的 8 个像素点计算梯度。

其中梯度计算是取 X 方向与 Y 方向的梯度，计算公式如下：

$$m(x,y) = \sqrt{(L(x+1,y) - L(x-1,y))^2 + (L(x,y+1) - L(x,y-1))^2}$$

角度计算公式为：

$$\theta(x,y) = \tan^{-1} \left(\frac{L(x,y+1) - L(x,y-1)}{L(x+1,y) - L(x-1,y)} \right)$$

梯度与角度计算完成后，对每个关键点周围的 8 个像素点计算梯度。然后根据 SIFT 算法计算梯度的公式，对每个关键点周围的 8 个像素点计算梯度。然后根据 SIFT 算法计算梯度的公式，对每个关键点周围的 8 个像素点计算梯度。

现大致步骤如下：

1) 输入关键点信息，初始化高斯函数。

```

// get current layer sigma
float octave_sigma = scaleOctaveSigma( 2.0f, 1.0f );
e001 = d[ ic ].data[ rac + xa ];
e101 = d[ ic ].data[ rac + xc ];
e201 = d[ ic ].data[ rac + xb ];

```

```

// create a circle of radius 3 pixels around the key point
float2DArray gaussianMask = ...
e011 = d[ ic ].data[ rc + xa ];
e111 = d[ ic ].data[ rc + xc ];
e211 = d[ ic ].data[ rc + xb ];

```

```

// create a circle of radius 3 pixels around the key point
float2DArray gaussianMask = ...
e021 = d[ ic ].data[ rbc + xa ];
e121 = d[ ic ].data[ rbc + xc ];
e221 = d[ ic ].data[ rbc + xb ];

```

```

    if ( Math.abs( ox ) > 0.5 || Math.abs( oy ) > 0.5 || Math.
        e002 = d[ ibc ].data[ rac + xa ];
        e102 = d[ ibc ].data[ rac + xc ];
        e202 = d[ ibc ].data[ rac + xb ];
        xc = ( int ) Math.round( ( float ) xc + ox );
        e012 = d[ ibc ].data[ rc + xa ];
        e112 = d[ ibc ].data[ rc + xc ];
        e212 = d[ ibc ].data[ rc + xb ];
        e022 = d[ ibc ].data[ rbc + xa ];
        e122 = d[ ibc ].data[ rbc + xc ];
        e222 = d[ ibc ].data[ rbc + xb ];
    }
    else
    {
        fx = ( float ) xc + ox;
        fy = ( float ) yc + oy;
        fi = ( float ) ic + oi;
        if ( fx < 0 || fy < 0 || fi < 0 || fx > d[ 0 ].width -
            -1 || fy > d[ 0 ].height - 1 || fi > d.length -
            1 )
        {
            isLocalizable = false;
        }
        else
        {
            isLocalized = true;
        }
    }
    else isLocalizable = false;
}
while ( !isLocalized && isLocalizable && t >= 0 );
// reject detections that could not be localized properly

if ( !isLocalized )
{
    continue;
}

// reject detections with very low contrast
if ( Math.abs( ell1 + 0.5f * ( dx * ox + dy * oy + di * oi ) ) < CONTRAST_
    THRESHOLD ) continue;

// reject edge responses by Harris
float det = dxx * dyy - dxy * dxy;

```

```
float trace = dxx + dyy;
if ( trace * trace / det > MAX_CURVATURE_RATIO ) continue;

candidates.addElement( new float[]{ fx, fy, fi } );
```

上述代码可以分为三个部分，第一部分是实现中心点周围 26 个像素点比较，第二部分是实现在三维空间的偏移计算与极大值定位，最后一部分实现极大值过滤，其中三维空间是指任意一个像素的位置都是由 (x, y, σ) 组成的。

3. 关键点方向指派

经过上面两步以后得到的关键点具备了放缩不变性的特征，因此，关键点方向指派主要实现关键点的旋转不变性。实现方向指派是通过对关键点周围圆形像素子区域内的像素计算梯度与角度，并通过角度直方图来寻找梯度变化最大的像素。建立直方图时，以每 10 度为一个 BIN 建立直方图，这样即使得到最大直方图，仍然无法确切知道角度具体的大小值。根据抛物线原理，如果知道它左右值，就可以通过插值得到精确的角度最大值，作为该关键点的角度值，这样就完成了关键点方向指派。

同时根据 SIFT 算法作者的论文^①可知，还应该对大于最大值 80% 的 BIN 做同样的计算，以作为关键点方向，统计证明约有 15% 的关键点有多个方向，这样做的好处是提供 SIFT 算法匹配与识别率，感兴趣的读者可以自己完成该部分的代码。

其中梯度计算是取 X 方向与 Y 方向的梯度值的平方根，公式表示为如下：

$$m(x, y) = \sqrt{(L(x+1, y) - L(x-1, y))^2 + (L(x, y+1) - L(x, y-1))^2}$$

角度计算公式为：

$$\theta(x, y) = \tan^{-1}((L(x, y+1) - L(x, y-1)) / (L(x+1, y) - L(x-1, y)))$$

梯度计算的代码实现被放在了工具类 GaussianUtil.java 中。完整的关键点方向指派代码实现大致步骤如下：

1) 输入关键点信息，初始化高斯核模板。

```
// get current layer sigma of the Octave
float octave_sigma = scaleOctave.sigma[ 0 ] * ( float )Math.pow( 2.0f, c[ 2 ] );

// create a circular gaussian window with sigma 1.5 times that of the key point
Float2DArray gaussianMask =
    GaussianUtil.create_gaussian_kernel_2D_offset(
        octave_sigma * 1.5f,
        c[ 0 ] - ( float )Math.floor( c[ 0 ] ), // x
```

① David G. Lowe – “Distinctive Image Features from Scale-Invariant Keypoints”, January 5, 2004.

```

        c[ 1 ] = ( float ) Math.floor( c[ 1 ] ), // y
        false );

```

2) 计算当前层图像每个像素数的梯度与角度，根据高斯模板大小得到关键点周围像素梯度与角度。

```

// get the gradients in a region around the key points location
Float2DArray[] src = GaussianUtil.createGradients(scaleOctave.getLevel(Math.
    round(c[ 2 ] )));
Float2DArray[] gradientROI = new Float2DArray[ 2 ];
gradientROI[ 0 ] = new Float2DArray( gaussianMask.width, gaussianMask.width );
gradientROI[ 1 ] = new Float2DArray( gaussianMask.width, gaussianMask.width );

int half_size = gaussianMask.width / 2;
int p = gaussianMask.width * gaussianMask.width - 1;
for ( int yi = gaussianMask.width - 1; yi >= 0; --yi )
{
    int ra_y = src[ 0 ].width * Math.max( 0, Math.min( src[ 0 ].height - 1, ( int )c[
        1 ] + yi - half_size ) );
    int ra_x = ra_y + Math.min( ( int )c[ 0 ], src[ 0 ].width - 1 );
    for ( int xi = gaussianMask.width - 1; xi >= 0; --xi )
    {
        int pt = Math.max( ra_y, Math.min( ra_y + src[ 0 ].width - 2, ra_x + xi -
            half_size ) );
        gradientROI[ 0 ].data[ p ] = src[ 0 ].data[ pt ];
        gradientROI[ 1 ].data[ p ] = src[ 1 ].data[ pt ];
        --p;
    }
}

```

3) 对关键点周围像素梯度完成高斯权重计算。

```

// calculate weighted gradient of each pixels around key points
for ( int i = 0; i < gradientROI[ 0 ].data.length; ++i )
{
    gradientROI[ 0 ].data[ i ] *= gaussianMask.data[ i ];
}

```

4) 建立角度直方图，获取最大角度。

```

// build an orientation histogram of the sub region
for ( int i = 0; i < gradientROI[ 0 ].data.length; ++i )
{
    int bin = Math.max( 0, ( int ) ( ( gradientROI[ 1 ].data[ i ] + Math.PI ) /
        ORIENTATION_BIN_SIZE ) );
    histogram_bins[ bin ] += gradientROI[ 0 ].data[ i ];
}

// find the max value
int max_i = 0;

```

```

for ( int i = 0; i < ORIENTATION_BINS; ++i )
{
    if ( histogram_bins[ i ] > histogram_bins[ max_i ] ) max_i = i;
}

```

5) 插值计算得到最终精确角度值。

```

float e0 = histogram_bins[ ( max_i + ORIENTATION_BINS - 1 ) % ORIENTATION_BINS ];
float e1 = histogram_bins[ max_i ];
float e2 = histogram_bins[ ( max_i + 1 ) % ORIENTATION_BINS ];
float offset = ( e0 - e2 ) / 2.0f / ( e0 - 2.0f * e1 + e2 );
float orientation = ( ( float )max_i + offset ) * ORIENTATION_BIN_SIZE - ( float )
    Math.PI;

```

4. 关键点描述子生成

对得到的每个关键点，取关键点在中心位置的 16×16 像素窗口，计算窗口内每个像素的角度与梯度，并且对得到的梯度值乘以高斯权重。将 16×16 的像素区域划分为 4×4 的 16 个子区域，每个区域 16 个像素，扫描子区域的每个像素，对每个子区域建立方向直方图，方向在 $0 \sim 360$ 度之间，且分为 8 个等份，每个等份方向上的值是各个子区域梯度值的累加，这样就得到了 $4 \times 4 \times 8 = 128$ 个关键点描述子。对全部关键点完成此操作，即得到图像 SIFT 特征提取结果。操作图如图 13-11 所示。

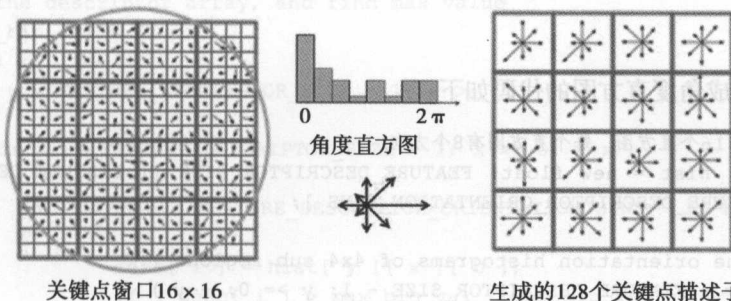


图 13-11 关键点描述子生成

采样生成 16×16 关键点窗口的代码如下：

```

// sample the region around the key points location
for ( int y = FEATURE_DESCRIPTOR_WIDTH - 1; y >= 0; --y )
{
    float ys =
        ( ( float )y - 2.0f * ( float )FEATURE_DESCRIPTOR_SIZE + 0.5f ) *
        octave_sigma; //!< scale y around 0,0
    for ( int x = FEATURE_DESCRIPTOR_WIDTH - 1; x >= 0; --x )
    {
        float xs =
            ( ( float )x - 2.0f * ( float )FEATURE_DESCRIPTOR_SIZE + 0.5f ) *
            octave_sigma; //!< scale x around 0,0
        float yr = cos_o * ys + sin_o * xs; //!< rotate y around 0,0
    }
}

```



```

float xr = cos_o * xs - sin_o * ys; //!< rotate x around 0,0

// translate ys to sample y position in the gradient image
int yg = GaussianUtil.flipInRange(
    (int)(Math.round( yr + c[ 1 ] )),
    gradients[ 0 ].height );

// translate xs to sample x position in the gradient image
int xg = GaussianUtil.flipInRange(
    (int)(Math.round( xr + c[ 0 ] )),
    gradients[ 0 ].width );

// get the samples
int region_p = FEATURE_DESCRIPTOR_WIDTH * y + x;
int gradient_p = gradients[ 0 ].width * yg + xg;

// weigh the gradients
region[ 0 ].data[ region_p ] = gradients[ 0 ].data[ gradient_p ] *
    descriptorMask[ y ][ x ];

// rotate the gradients orientation it with respect to the features
orientation
region[ 1 ].data[ region_p ] = gradients[ 1 ].data[ gradient_p ] -
    orientation;
}
}

```

基于窗口生成角度直方图的代码如下：

```

// 建立4x4的16个直方图，每个直方图有8个方向
float[][][] hist = new float[ FEATURE_DESCRIPTOR_SIZE ][ FEATURE_DESCRIPTOR_SIZE
    ][ FEATURE_DESCRIPTOR_ORIENTATION_BINS ];

// build the orientation histograms of 4x4 sub regions
for ( int y = FEATURE_DESCRIPTOR_SIZE - 1; y >= 0; --y )
{
    int yp = FEATURE_DESCRIPTOR_SIZE * 16 * y;
    for ( int x = FEATURE_DESCRIPTOR_SIZE - 1; x >= 0; --x )
    {
        int xp = 4 * x;
        for ( int ysr = 3; ysr >= 0; --ysr )
        {
            int ysrp = 4 * FEATURE_DESCRIPTOR_SIZE * ysr;
            for ( int xsr = 3; xsr >= 0; --xsr )
            {
                // make it scope in 0 ~ 2PI
                float bin_location = ( region[ 1 ].data[ yp + xp + ysrp
                    + xsr ] + ( float )Math.PI ) / ( float )FEATURE_
                    DESCRIPTOR_ORIENTATION_BIN_SIZE;
                // calculate rate for line nearest interpolation
                int bin_b = ( int )( bin_location );
            }
        }
    }
}

```

```

int bin_t = bin_b + 1;
float d = bin_location - ( float )bin_b;
// make it value in scope[0, 7],bad way!!,
// can check use if(bin_t > 7) bin_b = 6, bin_t = 7, ugly code!!
bin_b = ( bin_b + 2 * FEATURE_DESCRIPTOR_ORIENTATION_
    BINS ) % FEATURE_DESCRIPTOR_ORIENTATION_
    BINS;
bin_t = ( bin_t + 2 * FEATURE_DESCRIPTOR_ORIENTATION_
    BINS ) % FEATURE_DESCRIPTOR_ORIENTATION_
    BINS;
// 高斯权重梯度
float t = region[ 0 ].data[ yp + xp + ysrp + xsr ];
// 基于权重精准累加
hist[ y ][ x ][ bin_b ] += t * ( 1 - d );
hist[ y ][ x ][ bin_t ] += t * d;
    }
}
}

```

基于角度直方图结果，生成关键点 128 描述子的代码如下：

```

// define 128 descriptor
float[] desc = new float[ FEATURE_DESCRIPTOR_SIZE * FEATURE_DESCRIPTOR_SIZE *
    FEATURE_DESCRIPTOR_ORIENTATION_BINS ];

// build the descriptor array, and find max value
float max_bin_val = 0;
int i = 0;
for ( int y = FEATURE_DESCRIPTOR_SIZE - 1; y >= 0; --y )
{
    for ( int x = FEATURE_DESCRIPTOR_SIZE - 1; x >= 0; --x )
    {
        for ( int b = FEATURE_DESCRIPTOR_ORIENTATION_BINS - 1; b >= 0; --b )
        {
            desc[ i ] = hist[ y ][ x ][ b ];
            if ( desc[ i ] > max_bin_val ) max_bin_val = desc[ i ];
            ++i;
        }
    }
}

```

完整的关键点描述子生成代码参见 SIFTFeatureDetector 类中的 createDescriptor() 方法，其中还包括了最后一步对得到的描述子进行归一化。

5. SIFT 算法代码说明

完整的 SIFT 算法代码实现参见本书源文件的包 com.book.chapter.thirteen.sift 中所有代码类，SIFT 算法实现是本书前面所学知识的综合运用，涉及图像卷积、高斯模糊、图像像素归一化处理、计算图像像素梯度与角度、图像高斯金字塔建立、下采样、Harris 特征提取等。其中本人认为影响 SIFT 特征提取最关键的在于第二步，即极值寻找与定位、过滤，其中的插值方法是图像二维插值的延伸。再次强调一下源代码也是本书的一部分，阅读与运行本书中源

代码有助于读者加深对所学知识的理解，跨越图像处理理论与实践脱节的鸿沟，更好地理解所学知识。

13.8 小结

本章介绍了图像特征提取常见算法，从最简单的图像颜色特征提取开始，过渡到基于灰度共生矩阵的（GLCM）纹理提取，然后介绍了图像中最重要的变换——霍夫变换，并详细介绍了基于霍夫变换检测直线与圆的原理和代码实现步骤。此外，还基于前面学习到图像卷积知识，介绍了图像高斯金字塔提取过程，以及提取图像中边缘角度的算法，即 Harris 角度特征检测算法。最后在本书前面所学的知识基础上，着重介绍图像 SIFT 特征提取这一重要算法，通过详细剖析实现步骤与代码实现，帮助读者厘清 SIFT 算法如何实现尺度与旋转不变性特征提取，建立最终的 SIFT 特征描述子的整个过程。

在学习图像处理知识的同时，本章也涉及一些简单数学概念，比如矩阵的特征向量、矩阵反转、一些简单的高斯核生成、归一化处理等，这些在代码中均有体现，相信读者只要查阅相关知识，稍作了解，对照代码实现，不难掌握。在这里还想再次说明，图像特征提取涉及较多的数学知识，想要深入阅读与研究，具备这些基本的数学知识是必不可少的条件之一。本章内容只是管中窥豹，希望帮读者打开图像特征提取这扇门。

综合运用：照片转油画算法

从第 1 ~ 13 章，介绍了图像处理各种知识，本章将介绍一个当下十分流行的照片转手绘油画算法。想象一下，当画家以照片为模板开始创造一幅对应的油画时，首先画出来的会是什么？肯定是轮廓，然后才是细节，一笔一笔地画下去，最终得到完整的油画。而我们要做的就是让程序来模仿这一过程，其中最关键就是笔画（Stroke）生成。

该算法的基本原理基于一篇学术论文（Enhancement of Moment Based Painterly Rendering Using Connected Components），主要是通过计算像素的 Moments 得到笔画区域，然后通过灰度图像的抖动算法实现笔画位置采样，在得到位置以后通过计算区域 Moments 得到笔画的中心位置、宽、高、角度等参数，形成每个笔画，根据笔画大小，从最大笔画开始绘制，直到所有笔画绘制完成就得到了完整的油画图像。上述过程每步都会涉及前面章节中已经学到的一些图像处理知识，所以说本章可帮助大家加深对前面所学知识的理解与实战运用。

14.1 画笔区域

实现图像转手绘油画的第一步就是要对输入图像进行画笔区域（Stroke Area）处理，该步骤的主要原理是根据指定窗口大小，计算窗口内像素的 Moments 值作为该窗口内中心像素点的值，这里取 Moment 的零阶层值，图像 Moments 的计算公式如下：

$$M_{lm} = \sum_x \sum_y x^l y^m I(x, y)$$

因为输入的是彩色 RGB 图像，所以为了得到每个像素点只有一个值的灰度像素点，要先计算每个像素点 RGB 三通道值与中心像素点 RGB 之间的欧几里得距离，然后根据输入参数 $d02$ 处理得到 0 ~ 1 之间的比值。完整的画笔区域生成大致分为如下几步：

- 1) 初始化输入参数, 默认窗口大小 $size = 10$ 、 $d02 = 150 \times 150$ 。
- 2) 循环处理每个像素
 - a) 计算窗口内像素的 Moments 值。
 - b) 对 Moments 值结果除以窗口大小, 然后乘以 255, 得到 $0 \sim 255$ 之间的灰度值。
 - c) 作为该像素点的新值写到输出像素数组中。
- 3) 将输出的像素数组赋值到输出图像中, 返回笔画区域图像。

完整的实现类代码 `StrokeAreaFilter.java` 可以参见源文件中第 14 章相关部分, 核心部分代码如下所示:

□ 获取像素, 初始化窗口大小的代码片段如下:

```
int[] inPixels = new int[width*height];
int[] outPixels = new int[width*height];
getRGB( src, 0, 0, width, height, inPixels );
int index = 0, index2 = 0;
int semiRow = (int)(size/2);
int semiCol = (int)(size/2);
int newX, newY;
```

```
// initialize the color RGB array with zero...
```

```
int[] rgb = new int[3];
int[] rgb2 = new int[3];
for(int i=0; i<rgb.length; i++) {
    rgb[i] = rgb2[i] = 0;
}
```

□ 实现 Moment 计算得到每个像素计算结果的代码如下:

```
index = row * width + col;
ta = (inPixels[index] >> 24) & 0xff;
rgb[0] = (inPixels[index] >> 16) & 0xff;
rgb[1] = (inPixels[index] >> 8) & 0xff;
rgb[2] = inPixels[index] & 0xff;
```

```
/* adjust region to fit in source image */
```

```
// color difference and moment Image
```

```
double moment = 0.0d;
```

```
for(int subRow = -semiRow; subRow <= semiRow; subRow++) {
```

```
    for(int subCol = -semiCol; subCol <= semiCol; subCol++) {
```

```
        newY = row + subRow;
```

```
        newX = col + subCol;
```

```
        if(newY < 0) {
```

```
            newY = 0;
```

```
        }
```

```
        if(newX < 0) {
```

```
            newX = 0;
```

```
        }
```

```
        if(newY >= height) {
```

```
            newY = height-1;
```

```

    }
    if(newX >= width) {
        newX = width - 1;
    }
    index2 = newY * width + newX;
    rgb2[0] = (inPixels[index2] >> 16) & 0xff; // red
    rgb2[1] = (inPixels[index2] >> 8) & 0xff; // green
    rgb2[2] = inPixels[index2] & 0xff; // blue
    moment += colorDiff(rgb, rgb2);
}

// calculate the output pixel value.
int outPixelValue = clamp((int) (255.0d * moment / (size*size)));
outPixels[index] = (ta << 24) | (outPixelValue << 16) | (outPixelValue << 8) |
    outPixelValue;

```

□ 计算颜色距离的代码如下：

```

public static double colorDiff(int[] rgb1, int[] rgb2)
{
    // (1-(d/d0)^2)^2
    double d2, r2;
    d2 = colorDistance (rgb1, rgb2);

    if (d2 >= d02)
        return 0.0;

    r2 = d2 / d02;

    return ((1.0d - r2) * (1.0d - r2));
}

public static double colorDistance(int[] rgb1, int[] rgb2)
{
    int dr, dg, db;
    dr = rgb1[0] - rgb2[0];
    dg = rgb1[1] - rgb2[1];
    db = rgb1[2] - rgb2[2];
    return dr * dr + dg * dg + db * db;
}

```

测试与运行该代码时，可以运用本章提供的 MainUI.java 类，该用户界面提供了选择图像与处理图像两个功能，其中【Paint】按钮将触发照片转油画功能。具体如何运行测试 StrokeAreaFilter 类，可以参照本书第3章中测试类的例子，这里就不再赘述。

14.2 采样问题

从画笔区域图像可以得到那些需要着重绘制的区域，可是每个细节需要绘制多少笔画

呢？这样就转换为采样问题了，这里的采样算法采用了弗洛伊德-斯坦德伯格抖动算法，该算法是将灰度图像转为黑白图像时的经典采样算法，基于错误扩散完成。本书的第10章对该算法有详细描述与代码实现。注意，为了更好地体现采样的随机性，这里增加了错误扩散的随机性。在类 `StrokeGenerator` 的 `strokePosition` 方法中实现画笔区域抖动采样的代码如下：

```
int x, y;
float error, value;
float a = (float) ((s-1)/Math.pow(255.0, p));
Random rand = new Random();
// initialization the buffer rows
float[] currow = new float[width];
float[] nxtrow = new float[width];
int[] posArea = new int[width * height];
for (x = 0; x < width; x++)
{
    nxtrow[x] = Math.max(1.0f/(a*(float) (Math.pow(getGrayPixels(inpixels,
        x), p))+1), 0.5f);
}
// start to dither algorithm
for(int row=1; row<height-1; row++)
{
    /* next line becomes current line */
    swap(currow, nxtrow);
    /* copies next line to local buffer */
    nxtrow[0] = Math.max(1.0f/(a*(float) (Math.pow(getGrayPixels(inpixels, row *
        width), p))+1), 0.5f);
    for (x = 1; x < width; x++) {
        nxtrow[x] = 1.0f/(a*(float) (Math.pow(getGrayPixels(inpixels, (row *
            width + x)), p))+1);
    }
    /* spread error */
    for (x = 1; x < width-1; x++) {
        value = currow[x] > 1.0f ? 1.0f : 0.0f;
        error = currow[x] - value;
        int gray = value > 0.0 ? 0 : 255;
        posArea[row * width + x] = (255 << 24) | (gray << 16) | (gray << 8) | gray;
        switch (rand.nextInt(100) % 4) {
            case 0:
                nxtrow[x + 1] += error/16;
                nxtrow[x - 1] += 3*error/16;
                nxtrow[x] += 5*error/16;
                currow[x + 1] += 7*error/16;
                break;
            case 1:
                nxtrow[x + 1] += 7*error/16;
                nxtrow[x - 1] += error/16;
                nxtrow[x] += 3*error/16;
                currow[x + 1] += 5*error/16;
                break;
```

```

        case 2:
            nxtrow[x + 1] += 5*error/16;
            nxtrow[x - 1] += 7*error/16;
            nxtrow[x]      += error/16;
            currow[x + 1] += 3*error/16;
            break;
        case 3:
            nxtrow[x + 1] += 3*error/16;
            nxtrow[x - 1] += 5*error/16;
            nxtrow[x]      += 7*error/16;
            currow[x + 1] += error/16;
            break;
    }
}

// post process for last row pixels
for (x = 0; x < width; x++)
{
    int gray = nxtrow[x] > 1.0f ? 0 : 255;
    posArea[(height-1) * width + x] = (255 << 24) | (gray << 16) | (gray << 8) | gray;
}
return posArea;

```

其中 getGrayPixels 方法是获取对应像素点的灰度值, 代码如下:

```

private int getGrayPixels(int[] pixels, int index)
{
    int gray = (pixels[index] >> 16) & 0xff;
    return gray;
}

```

14.3 笔画参数

通过上面两步, 我们很顺利地找到了画笔区域与位置, 现在需要计算笔画参数 (Stroke Parameters), 即需要知道笔画宽度与高度、角度、中心位置等信息, 这样才能生成每个笔画。这些参数是对采样之后的图像中非白色背景像素所在窗口区域内所有像素 Moments 的零阶、一阶与二阶值计算后得到的。这些内容对读者来说并不陌生, 本书第 10 章详细介绍过如何通过 Moment 值计算得到图像质心和方向, 这里正好用上。计算采样点中心位置坐标点的公式如下:

$$x_c = \frac{M_{10}}{M_{00}} \quad y_c = \frac{M_{01}}{M_{00}}$$

角度 (θ)、宽度 (w) 与高度 (l) 计算公式如下:

$$\theta = \frac{\tan^{-1}\left(\frac{b}{a-c}\right)}{2} \quad w = \sqrt{6(a+c - \sqrt{b^2 + (a-c)^2})} \quad l = \sqrt{6(a+c + \sqrt{b^2 + (a-c)^2})}$$

其中 a、b、c 的值由如下公式得到:

$$a = \frac{M_{20}}{M_{00}} - x_c^2 \quad b = 2 \left(\frac{M_{11}}{M_{00}} - x_c y_c \right) \quad c = \frac{M_{02}}{M_{00}} - y_c^2$$

□ 完整的计算笔画参数, 生成笔画对象的代码如下:

```
// declare variables
double m00, m01, m10, m11, m02, m20;
double a, b, c;
double tempval;
double dw, dxc, dyc;

// calculate moments
int[] wh = new int[2]; // width, height
int[] roiArea = MomentsUtil.getRoi(scaledInput, x, y, s, s, sw, sh, wh);
int[] xyrgb = getColorPixels(scaledInput, (y * sw + x));
double[] mms = new double[6];
MomentsUtil.calculateMoments(roiArea, xyrgb, mms, wh[0], wh[1]);
m00 = mms[0];
m01 = mms[1];
m10 = mms[2];
m11 = mms[3];
m02 = mms[4];
m20 = mms[5];

// calculate parameters
dxc = m10 / m00;
dyc = m01 / m00;
a = (m20 / m00) - (double)((dxc)*(dxc));
b = 2 * (m11 / m00 - (double)((dxc)*(dyc)));
c = (m02 / m00) - (double)((dyc)*(dyc));
double theta = Math.atan2(b, (a-c)) / 2;
tempval = Math.sqrt(b*b + (a-c)*(a-c));
dw = Math.sqrt(6 * (a+c - tempval));
float w = (float)(Math.sqrt(6 * (a+c - tempval)));
float l = (float)(Math.sqrt(6 * (a+c + tempval)));
int xc = (int)(x + Math.floor(dxc - s/2));
int yc = (int)(y + Math.floor(dyc - s/2));
// factor*xc, factor*yc, factor*w, factor*l, (float) theta, rgb, level
StrokeElement element = new StrokeElement(factor*xc, factor*yc, factor*w,
factor*l, level, (float)theta, xyrgb);
return element;
```

□ 其中根据采样点与输入的窗口大小参数, 获得采样点周围 ROI 区域的代码如下:

```
public static int[] getRoi(int[] scaledInput, int xc, int yc, int width, int
height, int sw, int sh, int[] wh) {
int x0 = xc - width/2;
int y0 = yc - height/2;
/* adjust region to fit in source image */
if (x0 < 0) {
```

```

width += x0;
x0 = 0;
}
if (x0 > width) x0 = width;
if (y0 < 0) {
    height += y0;
    y0 = 0;
}
if (y0 > height) y0 = height;
if ((x0 + width) > sw) {
    width = sw - x0;
}
if ((y0 + height) > sh) {
    height = sh - y0;
}
wh[0] = width;
wh[1] = height;
int[] roi = new int[width * height];
int rr = 0;
int cc = 0;
for(int row=y0; row<(y0 + height); row++)
{
    cc = 0;
    for(int col= x0; col < (x0+width); col++)
    {
        int index = row * sw + col;
        roi[rr*width + cc] = scaledInput[index];
        cc++;
    }
    rr++;
}
return roi;
}

```

□ 计算得到 Moments 的零阶、一阶与二阶等六个初始值的代码如下：

```

int y, x;
int[] outPixels = colorDiff(roiArea, xyrgb, width, height);
Arrays.fill(mms, 0);
int index = 0;
// *m00 = *m01 = *m10 = *m11 = *m02 = *m20 = 0;

for (y = 0; y < height; y++) {
    for (x = 0; x < width; x++) {
        index = y*width + x;
        mms[0] += outPixels[index]; // m00
        mms[1] += y * (outPixels[index]); // m01;
        mms[2] += x * (outPixels[index]); // m10;
        mms[3] += y * x * (outPixels[index]); // m11;
        mms[4] += y * y * (outPixels[index]); // m02;
        mms[5] += x * x * (outPixels[index]); // m20;
    }
}

```

其中 a_i 的值为由下公式得到：

}

在处理 ROI 得到 Moments 从零到二各阶值之前，同样需要通过计算颜色之间距离，该处理与 StrokeAreaFilter 中的类似。这里不再赘述。对所有采样点完成上述计算就生成了对应采样图像所有的笔画信息，对原始图像在高与宽上缩小 $\frac{1}{2^n}$ （其中 n 为循环次数），分别实现笔画区域处理、采样与笔画的生成，这样做的目的是希望更多采样，使手绘油画图像看上去更加真实。该过程的代码实现如下：

```
int count = 0;
int factor = 1;
int level = 1;
int size = Math.max(width, height);
if (s == 0) {
    s = _DFS;
}
List<StrokeElement> allStrokes = new ArrayList<StrokeElement>();
StrokeGenerator sg = new StrokeGenerator();
while (size > 4 * s) {
    System.out.println("Processing level : " + factor);
    allStrokes.addAll(sg.getStrokes(inPixels, strokeArea, s, factor,
                                   level, width, height));
    size /= 2;
    factor *= 2;
    level++;
}
```

其中 s 是输入参数，决定笔画数采样的大小与多少，通常来说， s 值越小，生成的笔画数越多，得到油画图像与输入图像越相似。其中，StrokeElement 表示笔画信息的数据结构类。

14.4 笔画绘制

通过上面三小节的学习，我们可以顺利得到所有笔画，现在要解决的问题就是如何将这笔画绘制到一张空白的画布上，从而实现一幅真正的手绘油画。在根据笔画信息开始绘制之前，要解决三个问题：

- 1) 绘制顺序问题，笔画越大应该越先绘制，这符合绘画的一般认知。
- 2) 笔画模板问题，这里通过外部加载一个笔画图片解决。读者还可以自己创建笔画图片。
- 3) 由于加载的笔画大小是固定的，所以要根据生成的笔画信息进行适当放缩与旋转之后才能得到想要绘制的笔画。

这样三个问题都有了答案，现在就可以开始绘制了。绘制是基于透明度通道的画布像素与笔画信息中提供的像素进行对应位置的像素混合，关于像素混合在本书的第 5 章中有详细

介绍，不清楚的读者可以自己回顾一下。对所有笔画循环处理绘制以后，一幅手绘油画效果的图像就会生成。

□ 实现笔画大小排序的代码如下：

```
private void sortByDesc(List<StrokeElement> allStrokes) {
    int size = allStrokes.size();
    // selection sort
    for(int i=0; i<size-1; i++)
    {
        float d = allStrokes.get(i).getD();
        StrokeElement se = allStrokes.get(i);
        int selectedIndex = i;
        for(int j=i; j<size; j++)
        {
            float dd = allStrokes.get(j).getD();
            if(d <= dd)
            {
                // swap
                selectedIndex = j;
                d = dd;
            }
        }
        // swap it
        if(selectedIndex == i) continue;
        StrokeElement sej = allStrokes.get(selectedIndex);
        allStrokes.set(selectedIndex, se);
        allStrokes.set(i, sej);
    }
}
```

□ 根据笔画宽、高与角度实现笔画放缩与旋转的代码如下：

```
public BufferedImage getScaledImage(BufferedImage image, int w, int l) {
    BufferedImage scaledImage = new BufferedImage(w, l,
        BufferedImage.TYPE_INT_ARGB);
    Graphics2D graphics2D = scaledImage.createGraphics();
    graphics2D.setRenderingHint(RenderingHints.KEY_INTERPOLATION,
        RenderingHints.VALUE_INTERPOLATION_BILINEAR);
    graphics2D.drawImage(image, 0, 0, w, l, null);
    graphics2D.dispose();
    return scaledImage;
}

public BufferedImage rotateImage(BufferedImage image, double angle) {
    double sin = Math.abs(Math.sin(angle)), cos = Math.abs(Math.cos(angle));
    int w = image.getWidth(), h = image.getHeight();
    int neww = (int) Math.floor(w * cos + h * sin), newh = (int) Math
        .floor(h * cos + w * sin);
    GraphicsConfiguration gc = owner.getGraphicsConfiguration();
    BufferedImage result = gc.createCompatibleImage(neww, newh,
```



```

        Transparency.TRANSLUCENT);
        Graphics2D g = result.createGraphics();
        g.translate((neww - w) / 2, (newh - h) / 2);
        g.rotate(angle, w / 2, h / 2);
        g.drawRenderedImage(image, null);
        g.dispose();
        return result;
    }

```

□ 像素混合 imageBlend 方法的代码如下：

```

long la, li; /* line jumps for both images */
int xi, yi; /* lower left corner in input image */
int xa, ya; /* corresponding corner in alpha image */
int wa, ha; /* area in alpha image to be blended */
int x, y;
int r = rgb[0], g = rgb[1], b = rgb[2];

wa = stroke.getWidth();
xa = 0;
xi = xc - wa / 2;
if (xi < 0) {
    wa += xi;
    xa -= xi;
    xi = 0;
}
if (xi > width)
    return 0;
if (wa <= 0)
    return 0;
if (xi + wa >= width)
    wa = width - xi;

ha = stroke.getHeight();
ya = 0;
yi = yc - ha / 2;
if (yi < 0) {
    ha += yi;
    ya -= yi;
    yi = 0;
}
if (yi > height)
    return 0;
if (ha <= 0)
    return 0;
if (yi + ha >= height)
    ha = height - yi;

int[] sp = new int[stroke.getWidth() * stroke.getHeight()];
getRGB(stroke, 0, 0, stroke.getWidth(), stroke.getHeight(), sp);
getRGB(alpha, 0, 0, alpha.getWidth(), alpha.getHeight(), sp);

```

```

int index = 0;
for (y = 0; y < ha; y++) {
    for (x = 0; x < wa; x++) {
        index = y * stroke.getWidth() + x;
        int index2 = (y + yi) * width + (xi + x);
        int[] argb = getColorPixels(sp, index);
        int[] argb2 = getColorPixels(out, index2);
        float ba = argb[0] / 255.0f;
        float bai = 1.0f - ba;
        int ta = (int) (ba * argb[0] + bai * argb2[0]);
        int tr = (int) (ba * r + bai * argb2[1]);
        int tg = (int) (ba * g + bai * argb2[2]);
        int tb = (int) (ba * b + bai * argb2[3]);
        setColorPixels(out, index2, new int[] { ta, tr, tg, tb });
    }
}
return 1;

```

上述各个步骤合并起来, 实现的代码如下:

```

// sort stroke, from lager size to small size
sortByDesc(allStrokes);

// load stroke template
java.net.URL imageURL = this.getClass().getResource("stroke.png");
BufferedImage strokeTemplate = null;
try {
    strokeTemplate = ImageIO.read(imageURL);
} catch (IOException e) {
    System.err.println("An error occured when loading the image icon...");
}

// start to paint the stroke now!!!
BufferedImage canvasImage = new BufferedImage(width, height,
    BufferedImage.TYPE_INT_ARGB);
int[] resultPixels = new int[width * height];
// 白色背景画布
Arrays.fill(resultPixels, -1);
int totalStroke = 0;
for (StrokeElement element : allStrokes) {
    int sw = (int)element.getW();
    int sh = (int)element.getL();
    if(sw == 0 || sh == 0) continue;
    // 放缩
    BufferedImage scaledImage = getScaledImage(strokeTemplate, sh, sw);
    // 旋转
    BufferedImage rotatedImage = rotateImage(scaledImage, element.getTheta());
    // 绘制-像素混合
    imageBlend(resultPixels, width, height, rotatedImage, element.getRgb(),
        element.getXc(), element.getYc());
    totalStroke++;
}

```

完整油画绘制可以参考源文件中第 14 章的 `StrokePaintlyMain.java`。

14.5 程序实现

基于本章前四节所讲内容，完整实现从输入图像到输出油画图像的代码实现中，各类及相互关系说明如下：

- ❑ `MainUI` 类，实现用户界面接口，提供了选择图像与运行图像转油画显示功能。
- ❑ `MomentsUtil` 类，提供了计算图像 Moments 各阶功能的工具类。
- ❑ `ScaleFilter` 类，实现了图像放缩功能的滤镜类，在生成笔画信息时使用。
- ❑ `StrokeAreaFilter` 类，实现图像笔画区域提取功能。
- ❑ `StrokeElement` 类，笔画绘制时所需要信息数据结构类，用来存储笔画信息。
- ❑ `StrokeGenerator` 类，笔画生成类，实现了对笔画各种参数的计算。
- ❑ `StrokePaintlyMain` 类，使用上述各个类，根据输入图像，实现油画图像生成。

前四节已经介绍了上面非 UI 类的基本功能与代码实现，本节主要围绕 UI 类串联起整个程序功能实现。回顾本书前 3 章的知识，通过对 Swing 按钮组件添加事件监听，实现用户事件响应，在相应的响应方法中添加调用 `StrokePaintlyMain` 类代码即可实现图像转油画功能，当然前提是我们已经选择了一张图像。然后通过重载 `JComponent` 的 `paintComponent` 方法实现界面刷新，从而显示输出油画。`MainUI` 类正是用来实现上述功能的，其全部实现代码如下：

```
package com.book.chapter.fourteen;

import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.FlowLayout;
import java.awt.Graphics;
import java.awt.Toolkit;
import java.awt.Window;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;

import javax.imageio.ImageIO;
import javax.swing.JButton;
import javax.swing.JComponent;
import javax.swing.JFileChooser;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class MainUI extends JFrame implements ActionListener {
    /**
     *
     */
}
```

```

private static final long serialVersionUID = 3570033620825245822L;
public static final String PAINT_CMD = "Paint";
public static final String SELECT_CMD = "Select Image...";
private JButton paintBtn;
private JButton selectBtn;
private BufferedImage srcImage;
private BufferedImage destImage;
private JComponent imagePanel;
public MainUI()
{
    super("Automatic Paintly Render - GloomyFish");
    initComponents();
}

private void initComponents() {
    imagePanel = new JComponent()
    {
        /**
         *
         */
        private static final long serialVersionUID = 1L;

        @Override
        protected void paintComponent(Graphics g) {
            g.clearRect(0, 0, getWidth(), getHeight());
            if(srcImage != null)
            {
                g.drawImage(srcImage, 0, 0, srcImage.getWidth(),
                    srcImage.getHeight(), null);
            }
            if(destImage != null && srcImage != null)
            {
                g.drawImage(destImage, srcImage.getWidth() +
                    10, 0, destImage.getWidth(), destImage.
                    getHeight(), null);
            }
            if(srcImage == null && destImage == null)
            {
                g.drawString("Please select your image...", 100,
                    200);
            }
        }
    };
    this.getContentPane().setLayout(new BorderLayout());
    this.getContentPane().add(imagePanel, BorderLayout.CENTER);
    paintBtn = new JButton(PAINT_CMD);
    selectBtn = new JButton(SELECT_CMD);
    JPanel btnPanel = new JPanel();
    btnPanel.setLayout(new FlowLayout(FlowLayout.RIGHT));
    btnPanel.add(selectBtn);
    btnPanel.add(paintBtn);
}

```

```

        this.getContentPane().add(btnPanel, BorderLayout.SOUTH);
        selectBtn.addActionListener(this);
        paintBtn.addActionListener(this);
    }

    public void openView()
    {
        this.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        java.net.URL imageURL = this.getClass().getResource("rainbow-fish-md.png");
        try {
            setIconImage(ImageIO.read(imageURL));
        } catch (IOException e) {
            System.err.println("An error occurred when loading the image icon...");
        }
        this.setPreferredSize(new Dimension(800, 660));
        pack();
        centreView(this);
        setVisible(true);
    }

    public static void centreView(Window w) {
        Dimension me = w.getSize();
        Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
        int newX = (screenSize.width - me.width)/2;
        int newY = (screenSize.height - me.height)/2;
        w.setLocation(newX, newY);
    }

    public static void main(String[] args)
    {
        MainUI ui = new MainUI();
        ui.openView();
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        String command = e.getActionCommand();
        System.out.println("Command : " + command);
        if(command.equals(SELECT_CMD))
        {
            JFileChooser chooser = new JFileChooser();
            chooser.showOpenDialog(null);
            File f = chooser.getSelectedFile();
            if(f == null) return;
            try {
                srcImage = ImageIO.read(f);
            } catch (IOException e1) {
                e1.printStackTrace();
            }
        }
    }

```

```

        this.repaint();
    }
    else if(command.equals(PAINT_CMD))
    {
        // stroke area
        StrokePaintlyMain spm = new StrokePaintlyMain(this);
        destImage = spm.filter(srcImage, null);
        this.repaint();
    }
}
}

```

关于 Java Swing 的更多讨论已经超出本书范畴,感兴趣的读者可以阅读 JDK 官方文档中的 Swing 教程进行学习。本章所有 Java 类的源代码可以参考本书源文件中的第 14 章。

至此,基于 Moments 完整的图像转油画算法原理与实现就介绍完了。本例学习完后,希望读者能够建立起图像处理不是简单的知识点运用,而是一系列知识点的综合运用的概念。这里还想特别强调一下,源代码是本书的一部分,阅读、运行与调试本书中所有源代码有助于读者更好理解与掌握本书所介绍的知识。

14.6 小结

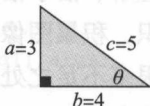
本章通过介绍一篇论文上有趣的油画算法实现,帮助读者灵活运用本书所学的各种图像知识,从而达到对本书所学知识的加深理解、活学活用,同时也希望能帮助读者掌握更多的图像处理编程技巧,掌握更多的实践知识,积累图像处理编程经验,提升对学习图像处理的兴趣与信心。当然本章实现的程序还有很多不足之处,首先是不能灵活调整输入参数,其次是像素混合做得不够好,希望感兴趣的读者能够进一步改进、完善。本章也是本书的最后一章,但是作者真诚希望本书能够成为各位学习图像处理的良好开端,而不是结束。

数学知识参考引用

1. 三角函数与反三角函数

三角函数在图像几何变换中的像素点坐标转换，常见的三角函数正弦、余弦、正切、余切公式如下：

$$\sin\theta = \frac{a}{c} = \frac{3}{5}, \cos\theta = \frac{b}{c} = \frac{4}{5}, \tan\theta = \frac{a}{b} = \frac{3}{4}, \cot\theta = \frac{b}{a} = \frac{4}{3}$$



图像处理中最常见的反三角函数是正切反三角函数，公式为 $\theta = \arctan x = \tan^{-1}x$ ，其取值范围为 $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$ ，需要注意的是本书中基于 Java 语言实现的反正切三角函数有两个 API，其中支持两个参数的正切反三角函数 API-Math.atan(y, x)，其取值范围为 $[-\pi, \pi]$ 。

2. 距离计算公式

平面坐标两点 $A(x_1, y_1)$ 与 $B(x_2, y_2)$ 之间的距离公式为：

$$|AB| = \sqrt{(y_2 - y_1)^2 + (x_2 - x_1)^2} \quad (1)$$

三维坐标两点 $A(x_1, y_1, z_1)$ 与 $B(x_2, y_2, z_2)$ 之间的距离公式为：

$$|AB| = \sqrt{(z_2 - z_1)^2 + (y_2 - y_1)^2 + (x_2 - x_1)^2} \quad (2)$$

在图像处理中 (1) 通常用于计算两个像素点之间的空间距离，(2) 通常用于计算两个像素点之间的 RGB 像素差异。

3. 矩阵运算

a) 矩阵加减要求两个矩阵的大小完全相同

假设矩阵 $A = \begin{bmatrix} 2 & 4 \\ 3 & 5 \end{bmatrix}$, 矩阵 $B = \begin{bmatrix} 1 & -2 \\ 4 & -3 \end{bmatrix}$

矩阵 $A+B$ 结果为:

$$C = \begin{bmatrix} 2+1 & 4+(-2) \\ 3+4 & 5+(-3) \end{bmatrix} = \begin{bmatrix} 3 & 2 \\ 7 & 2 \end{bmatrix}$$

矩阵 $A-B$ 结果为:

$$\text{矩阵 } C = \begin{bmatrix} 2-1 & 4-(-2) \\ 3-4 & 5-(-3) \end{bmatrix} = \begin{bmatrix} 1 & 6 \\ -1 & 8 \end{bmatrix}$$

b) 矩阵乘法, 假设矩阵 A 大小为 $m \times n$, 矩阵 B 大小为 $n \times p$, 则得到的最终矩阵 C 大小为 $m \times p$

矩阵 $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$ 大小为 3×2 , 矩阵 $B = \begin{bmatrix} 3 & 2 & 1 \\ 1 & -1 & 3 \end{bmatrix}$ 大小为 2×3

则矩阵 $C=AB$ 的大小为 3×3 , 结果为:

$$C = \begin{bmatrix} 3 \times 1 + 2 \times 1 & 1 \times 2 + 2 \times (-1) & 1 \times 1 + 2 \times 3 \\ 3 \times 3 + 4 \times 1 & 3 \times 2 + 4 \times (-1) & 3 \times 1 + 4 \times 3 \\ 5 \times 3 + 6 \times 1 & 5 \times 2 + 6 \times (-1) & 5 \times 1 + 6 \times 3 \end{bmatrix} = \begin{bmatrix} 5 & 0 & 7 \\ 13 & 2 & 15 \\ 21 & 4 & 23 \end{bmatrix}$$

c) 矩阵逆运算。简单举例, 假设矩阵 $A = \begin{bmatrix} 1 & -2 \\ 3 & 4 \end{bmatrix}$, 则其伴随矩阵 $A^* = \begin{bmatrix} 4 & 2 \\ -3 & 1 \end{bmatrix}$, 逆

$$\text{运算 } A^{-1} = \frac{A^*}{|A|} = \begin{bmatrix} \frac{4}{10} & \frac{2}{10} \\ \frac{-3}{10} & \frac{1}{10} \end{bmatrix}$$

4. 向量特征值

假设矩阵 $A = \begin{bmatrix} 2 & 7 \\ -1 & -6 \end{bmatrix}$, 计算特征值过程如下:

$$A - \lambda I = \begin{bmatrix} 2 & 7 \\ -1 & -6 \end{bmatrix} - \lambda \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 2-\lambda & 7 \\ -1 & -6-\lambda \end{bmatrix}$$

$$\det(A - \lambda I) = (2 - \lambda)(-6 - \lambda) + 7 = \lambda^2 + 4\lambda - 5 = (\lambda + 5)(\lambda - 1) = 0$$

即特征值 $\lambda_1 = -5$, $\lambda_2 = 1$ 。

5. 二次或三次多项式

$$y = ax^2 + bx + c$$

$$y = ax^3 + bx^2 + cx + d$$

主要用于像素插值计算。

6. 平面坐标与极坐标

假设平面坐标点 $P(x, y)$, 其中 $x = 12$, $y = 5$, 则它的极坐标对应点 $P(r, \theta)$, 其中 $r = \sqrt{x^2 + y^2} =$

13, 又 $\tan \theta = \frac{y}{x} = \frac{5}{12}$, 得到 $\theta = \tan^{-1}\left(\frac{5}{12}\right) = 22.6^\circ$ 。

7. 卷积 (定义与公式)

设: $f(x)$ 、 $g(x)$ 是 $R1$ 上的两个可积函数, 在有限范围 $[0, t]$ 内的卷积为:

$$[f * g](t) \equiv \int_0^t f(\tau)g(t - \tau) d\tau$$

当取值范围为 $[-\infty, \infty]$ 时:

$$f * g \equiv \int_{-\infty}^{\infty} f(\tau)g(t - \tau) d\tau = \int_{-\infty}^{\infty} g(\tau)f(t - \tau) d\tau$$

8. 高斯公式

一维高斯函数: $f(x) = \frac{1}{\sigma \sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$

二维高斯函数: $f(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{[(x-\mu_x)^2 + (y-\mu_y)^2]}{2\sigma^2}}$, 这里假设 $(\sigma_x = \sigma_y = \sigma)$

9. 导数

导数 (Derivative) 是微积分中的重要基础概念。当函数 $y = f(x)$ 的自变量 x 在一点 x_0 上产生一个增量 Δx 时, 如果存在函数输出值的增量 Δy 与自变量增量 Δx 的比值在 Δx 趋于 0 时的极限 a , a 即为在 x_0 处的导数, 记作 $f'(x_0)$ 或 $\frac{df}{dx}(x_0)$, 导数是函数的局部性质。一个函数在某一点的导数描述了这个函数在这一点附近的变化率, 所以常常在图像处理中用来计算图像像素值的梯度变化率。

假设有函数为: $f(x) = 3x^3 - 6x^2 + 2x - 1$

对应的一阶导数为: $f'(x) = \frac{df}{dx} = 9x^2 - 12x + 2$

对应的二阶导数为: $f''(x) = \frac{d^2f}{dx^2} = 18x - 12$

二阶导数的意义在于告诉我们一阶导数变化方向是增大还是减小。

10. 统计学 (计算数据均值、方差、标准偏差)

假设有采样数据 $a_0, a_1, a_2, a_3, a_4, \dots, a_{n-1}$, 则

$$\text{均值 } \mu = \frac{\sum_{i=0}^{n-1} a_i}{n}, \text{ 方差 } \sigma^2 = \frac{\sum_{i=1}^N (x_i - \mu)^2}{N}, \text{ 标准差 } \sigma = \sqrt{\frac{1}{N} \sum_{i=1}^N (x_i - \mu)^2}$$

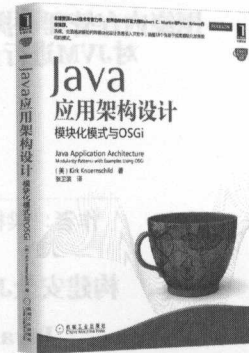
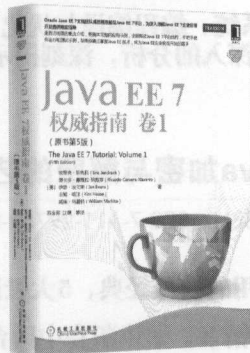
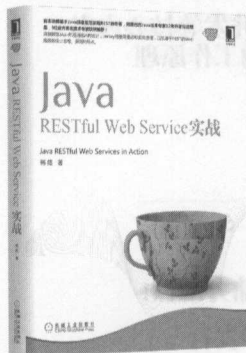
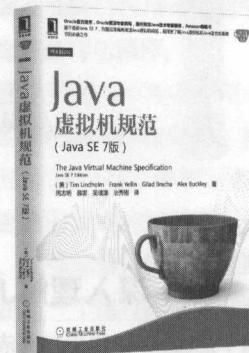
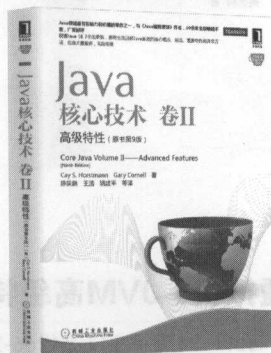
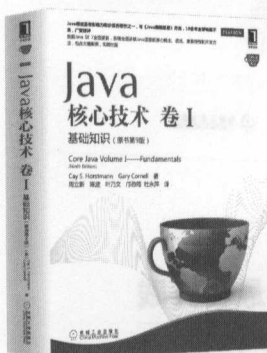
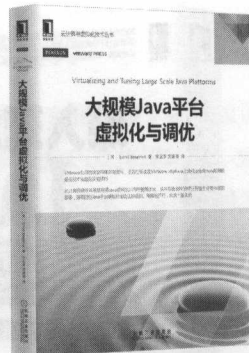
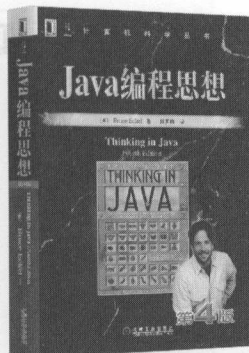
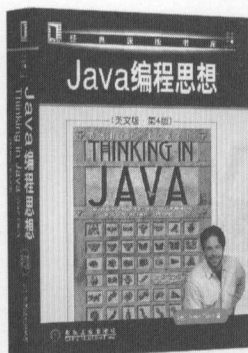
11. 相关数学知识学习推荐站点

<http://www.coolmath.com>

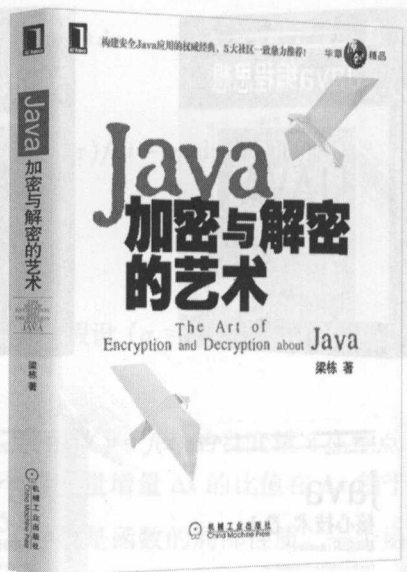
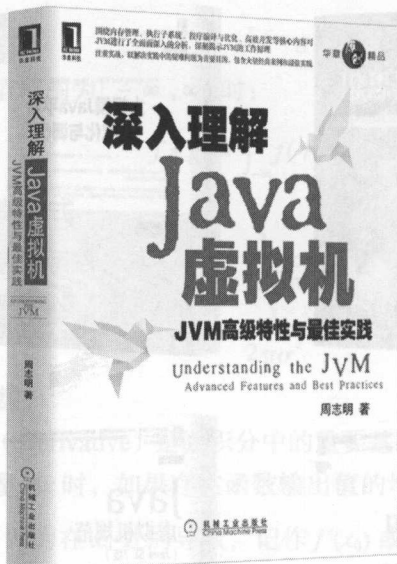
<http://mathworld.wolfram.com>

<http://www.mathsisfun.com>

推荐阅读



推荐阅读



深入理解Java虚拟机：JVM高级特性与最佳实践

作者：周志明 著 ISBN：978-7-111-34966-2 定价：69.00元

Java领域超级畅销书，9个月7次印刷，繁体版即将在中国台湾发行

围绕内存管理、执行子系统、编程编译与优化、高效并发等核心内容
对JVM进行全面而深入的分析，深刻揭示JVM的工作原理

Java加密与解密的艺术

作者：梁栋 著 ISBN：978-7-111-29762-8 定价：69.00元

构建安全Java应用的权威经典，5大社区一致鼎力推荐！

Java领域畅销书，繁体版在中国台湾同步发行

Java安全领域的百科全书和权威经典，开发企业级Java应用的必备参考手册

内容简介

本书全面介绍了各种常见图像处理知识的编程实现技巧及应用场景，跳出传统图像处理图书以数学推导、理论说明为主的方式，以编程实践来介绍所包含的各个知识点。

全书共分两个部分，基础篇（第1章~第3章）将简单地介绍Java Swing图形与图像编程基本API的使用技巧和相关编程实践，帮助读者了解图像接口在Java语言中的基础知识，并熟悉像素读写与操作。

图像处理与应用篇（第4章~第14章）则从最基础的像素操作开始，通过编程介绍图像处理所涉及的基本像素运算、混合、图像插值、直方图获取与图像搜索、图像卷积、边缘提取、二值图像分析与特征提取等知识，最后通过剖析一个流行的图像油画转换算法编程实践来结束本书。全面阐述了图像处理的理论知识与实践技巧，介绍了各知识点的编程实现思路与步骤。

Java

数字图像处理

编程技巧与应用实践

图像处理一直是计算机科学的热门研究课题，该领域涌现出了一大批杰出的图像处理人才，推动了社会的文明与进步，无论是医疗科技、农业科技、工业设计，还是检测、遥感测绘等，都受益于计算机图像处理技术的进步与发展。二十一世纪以来，伴随着互联网产业的发展与社会需求的推动，图像处理技术更是得到飞速发展，因此对图像处理专业人才的需求也更加迫切。然而由于图像处理是一门涉及众多学科知识，具备一定学习门槛的专业领域，因此让很多人望而生畏、裹足不前。本书将从最基础的图像处理概念入手，以代码驱动学习，带领大家一步一步踏入图像处理领域，书中注重理论与实践相结合，坚持以代码说话，用实践来加深对理论知识的认知与理解，真正做到授人以渔。

本书的主要内容和特色：

- 以程序员的独特视角阐述图像处理的每个知识点，尽量少引入复杂的数学公式推导，坚持以代码实践来演示理论。
- 通过实例剖析来配合理论知识的讲解，每一个章节都有对应知识点的代码实践，同时还有实现思路的讲解与步骤介绍，让读者能真正学以致用。
- 基于Java语言实现全部算法代码，不仅介绍了卷积、二值化处理、图像特征提取、分割等常见算法，还联系实际应用介绍了图像搜索、SIFT特征提取、基于Moment的油画算法等综合知识，以便更好地帮助读者掌握各种算法的编程实现技巧。



投稿热线: (010) 88379604
客服热线: (010) 88379426 88361066
购书热线: (010) 68326294 88379649 68995259

华章网站: www.hzbook.com
网上购书: www.china-pub.com
数字阅读: www.hzmedia.com.cn

上架指导: 计算机\图形图像

ISBN 978-7-111-51946-1



9 787111 519461 >

定价: 69.00元